华清远见
HQYJ.COM

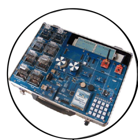
从实践中学嵌入式 Linux 操作系统

第 1 章 嵌入式 Linux 操作系统简介

在所有的操作系统中，Linux 是一个发展最快、应用最为广泛的操作系统，其本身的种种特性使其成为嵌入式开发中的首选。在进入市场的初期，嵌入式 Linux 设计通过广泛应用获得了巨大的成功。随着嵌入式 Linux 的成熟，它提供更小的尺寸和更多类型的处理器支持，并从早期的试用阶段迈入嵌入式的主流，抓住了电子消费类设备的开发者们的想象力。

华清远见





1.1 操作系统



操作系统（Operating System，OS）是电子计算机系统中负责支撑应用程序运行环境及用户操作环境的系统软件，同时也是计算机系统的核心与基石。它的职责包括对硬件的直接监管、对各种计算资源（如内存、处理器时间等）的管理，以及提供诸如作业管理之类的面向应用程序的服务等。

根据操作系统在用户界面的使用环境和功能特征的不同，操作系统一般可分为 3 种基本类型，即批处理系统、分时系统和实时系统。随着计算机体系结构的发展，又出现了许多种操作系统，包括嵌入式操作系统、个人操作系统、网络操作系统和分布式操作系统等。

1．批处理操作系统

批处理（Batch Processing）操作系统的工作方式是：用户将作业交给系统操作员，系统操作员将许多用户的作业组成一批作业，之后输入到计算机中，在系统中形成一个自动转接的连续的作业流；然后启动操作系统，系统自动、依次执行每个作业；最后由操作员将作业结果交给用户。

批处理操作系统的特点是：多道和成批处理。

2．分时操作系统

分时（Time Sharing）操作系统的工作方式是：一台主机连接了若干个终端，每个终端有一个用户在使用。用户交互式地向系统提出命令请求，系统接受每个用户的命令，采用时间片轮转方式处理服务请求，并通过交互方式在终端上向用户显示结果，用户根据上一步结果发出下一条命。分时操作系统将 CPU 的时间划分成若干个片段，称为时间片。操作系统以时间片为单位，轮流为每个终端用户服务。每个用户轮流使用一个时间片而使每个用户并不感到有别的用户存在。

分时系统具有多路性、交互性、“独占”性和及时性的特征。多路性，是指同时有多个用户使用一台计算机，宏观上看是多个人同时使用一个 CPU，微观上是多个人在不同时刻轮流使用 CPU；交互性，是指用户根据系统响应结果进一步提出新请求（用户直接干预每一步）；“独占”性，是指用户感觉不到计算机为其他人服务，就像整个系统为他所独占；及时性，是指系统对用户提出的请求及时响应。

常见的通用操作系统是分时系统与批处理系统的结合。其原则是：分时优先，批处理在后。“前台”响应需频繁交互的作业，如终端的要求；“后台”处理时间性要求不强的作业。

华清远见

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

3．实时操作系统

实时操作系统（Real Time Operating System，RTOS）是指使计算机能及时响应外部事件的请求，在规定的时间内完成对该事件的处理，并控制所有实时设备和实时任务协调、一致地工作的操作系统。实时操作系统追求的目标是：对外部请求在规定的时间范围内做出响应，有高可靠性和完整性。

4．嵌入式操作系统

嵌入式操作系统（Embedded Operating System）是运行在嵌入式系统环境中，对整个嵌入式系统及其所操作、控制的各种部件装置等资源进行统一协调、调度、指挥和控制的系统软件。

5．个人计算机操作系统

个人计算机操作系统是一种单用户多任务的操作系统，主要供个人使用，功能强、价格便宜，几乎可以在任何地方安装使用。它能满足一般人操作、学习、游戏等方面的需求。个人计算机操作系统的主要特点是：计算机在某一时间内为单个用户服务；采用图形界面人机交互的工作方式，界面友好；使用方便，用户无须专门学习，也能熟练操作机器。

6．网络操作系统

网络操作系统是基于计算机网络的，是在各种计算机操作系统上按网络体系结构协议标准开发的软件，包括网络管理、通信安全、资源共享和各种网络应用。其目标是相互通信及资源共享。

7．分布式操作系统

大量的计算机通过网络被连接在一起，可以获得极高的运算能力及广泛的数据共享，这种操作系统称为分布式操作系统（Distributed System）。

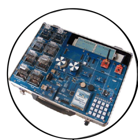
操作系统的主要功能简单总结为：操作系统位于底层硬件与用户之间，是两者沟通的桥梁。用户可以通过操作系统的用户界面输入命令，操作系统则对命令进行解释，驱动硬件设备，实现用户要求。

1.2 嵌入式系统



嵌入式系统是以应用为中心，以计算机技术为基础，软硬件可裁剪，适用于应用系统，对功能、可靠性、成本、体积、功耗等方面有特殊要求的专用计算机系统。

华清远见



嵌入式系统与通用计算机系统的本质区别在于系统应用不同，嵌入式系统是将一个计算机系统嵌入到对象系统中，这个对象可能是庞大的机器，也可能是小巧的手持设备，用户并不关心这个计算机系统的存在。

嵌入式系统一般包含嵌入式微处理器、外围硬件设备、嵌入式操作系统和应用程序 4 个部分。嵌入式领域已经有丰富的软硬件资源可以选择，涵盖通信、网络、工业控制、消费电子、汽车电子等各种行业。

嵌入式计算机系统与通用计算机系统相比具有以下特点。

- 嵌入式系统是面向特定系统应用的。嵌入式处理器大多数是专门为特定应用设计的，具有低功耗、体积小、集成度高等特点，一般是包含各种外围设备接口的片上系统。
- 嵌入式系统涉及计算机技术、微电子技术、电子技术、通信和软件等各行各业。它是一个技术密集、资金密集、高度分散、不断创新的知识集成系统。
- 嵌入式系统的硬件和软件都必须具备高度可定制性，只有这样才能适应嵌入式系统应用的需要，在产品价格性能等方面具备竞争力。
- 嵌入式系统的生命周期相当长。当嵌入式系统应用到产品以后，还可以进行软件升级，它的生命周期与产品的生命周期几乎一样长。
- 嵌入式系统不具备本地系统开发能力，通常需要有一套专门的开发工具和环境。

在计算机后 PC 技术时代，嵌入式系统将拥有最大的市场。计算机和网络已经全面渗透到日常生活的每一个角落，各种各样的新型嵌入式系统设备在应用数量上已经远远超过通用计算机，任何一个普通人可能拥有从大到小的各种使用嵌入式技术的电子产品，小到 MP3、PDA 等微型数字化产品，大到网络家电、智能家电、车载电子设备。而在工业和服务领域中，使用嵌入式技术的数字机床、智能工具、工业机器人、服务机器人也将逐渐改变传统的工业和服务方式。

美国著名的未来学家尼葛洛庞帝在 1999 年访华时曾预言，四五年后嵌入式系统将是继 PC 和 Internet 之后最伟大的发明。这个预言已经成为现实，现在的嵌入式系统正处于高速发展阶段。

1.3 嵌入式操作系统



嵌入式操作系统的一个重要特性是实时性。所谓实时性，就是在确定的时间范围内响应某个事件的特性。操作系统的实时性在某些领域是至关重要的，如工业控制、航空航天等领域。想象飞机正在空中飞行，如果嵌入式系统不能及时响应飞行员的控制指令，那么极有可能导致空难事故。有些嵌入式系统应用并不需要绝对的实时性，如 PDA 播放音乐，个别音频数据丢失并不影响效果。这可以使用软实时的概念来衡量。

华清远见 HQYJ.COM

据调查，目前全世界的嵌入式操作系统已经有两百多种。从 20 世纪 80 年代开始，出现了一些商用嵌入式操作系统，它们大部分是为专有系统而开发的。随着嵌入式领域的发展，各种各样的嵌入式操作系统相继问世，有许多商业的嵌入式操作系统，也有大量开放源代码的嵌入式操作系统，其中著名的嵌入式操作系统有：μC/OS、VxWorks、Neculeus、Linux 和 Windows CE 等。下面介绍一些主流的嵌入式操作系统。

1．Linux

Linux 操作系统是 UNIX 操作系统的一种克隆系统。它诞生于 1991 年的 10 月 5 日（这是第一次正式向外公布的时间）。以后借助于 Internet 网，并经过全世界各地计算机爱好者的共同努力下，现已成为今天世界上使用最多的一种 UNIX 类操作系统，并且使用人数还在迅猛增长。如图 1.1 所示是业内人士对国内 Linux 软件市场的预测。

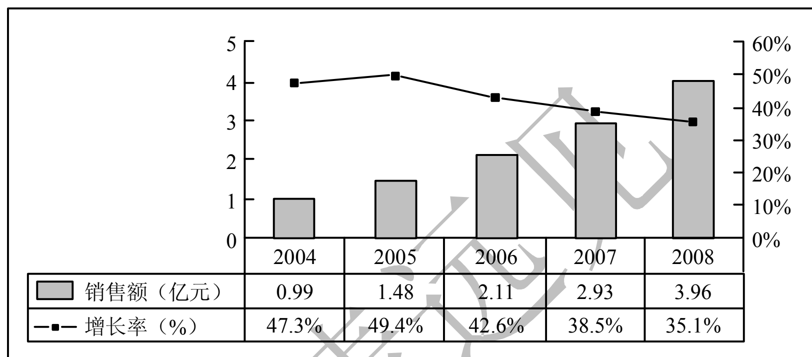


图 1.1　2004～2008 年国内 Linux 软件市场总量预测

根据 IDC 的报告，Linux 已经成为全球第二大操作系统。预计在服务器市场上，Linux 在未来几年内将以每年 25%的速度增长，中国的 Linux 市场更是保持 40%左右的增长速度。而在 Linux 操作系统方面，IDC 对中国在 2001～2006 年的市场预测发现，其市场占有率从 2001 年的 4.47%平稳地上升到 2006 年的 26.77%。

嵌入式 Linux 版本还有多种变体，如 RTLinux 通过改造内核实现了实时的 Linux；RTAI、Kurt 和 Linux/RK 也提供了实时能力；μCLinux 去掉了 Linux 的 MMU（内存管理单元），能够支持没有 MMU 的处理器等。

2．μC/OS

μC/OS 是一个典型的实时操作系统，该系统从 1992 年开始发展，目前流行的是第 2 个版本，即μC/OS-II。它的特点是：公开源代码，代码结构清晰，注释详尽，组织有条理，可移植性好；可裁剪，可固化；抢占式内核，最多可以管理 60 个任务。自从清华大学邵贝贝教授将 Jean J. Labrosse 的“μC/OS-Ⅱ：the Real Time Kernel”翻译后，在国内掀起了μC/OS-II 的热潮，特别是在教育研究领域。该系统短小精悍，是研究和学习实时操作系统的首选。

华清远见
HQYJ.COM
华清远见教育集团官网：www.hqyj.com

3. Windows CE

Windows CE 是微软的产品，它是从整体上为有限资源的平台设计的多线程、完整优先权、多任务的操作系统。Windows CE 采用模块化设计，并允许它对于从掌上电脑到专用的工控电子设备进行定制。操作系统的基本内核需要至少 200KB 的 ROM。从 SEGA 的 DreamCast 游戏机到现在大部分的高价掌上电脑都采用了 Windows CE。

随着嵌入式操作系统领域的竞争日益激烈，微软不得不应付来自 Linux 等免费系统的冲击。微软在 Windows CE.Net 4.2 版中，将增加一项授权价仅 3 美元的精简版本 WinCE.Net Core。WinCE.Net Core 具有基本的功能，包括实时 OS 核心（Real Time OS Kernel）、档案系统；IPv4、IPv6、WLAN、蓝牙等联网功能；Windows Media Codec；.Net 开发框架及 SQL Server.ce。微软推出低价版本 WinCE.Net，主要是看好语音电话、WLAN 的无线桥接器和个性化视听设备的市场潜力。

4. VxWorks

VxWorks 是 WindRiver 公司专门为实时嵌入式系统设计开发的操作系统软件，为程序员提供了高效的实时任务调度、中断管理，实时的系统资源及实时的任务间通信。应用程序员可以将尽可能多的精力放在应用程序本身，而不必再去关心系统资源的管理。该系统主要应用在单板机、数据网络（以太网交换机、路由器）和通信等多方面。其核心功能主要有以下几方面：

- 微内核 wind。
- 任务间通信机制。
- 网络支持。
- 文件系统和 I/O 管理。
- POSIX 标准实时扩展。
- C++及其他标准支持。

这些核心功能可以与 WindRiver 系统的其他附件和 Tornado 合作伙伴的产品结合在一起使用。谁都不能否认这是一个非常优秀的实时系统，但其昂贵的价格使不少厂商望而却步。

5. QNX

QNX 是一款实时操作系统，由加拿大 QNX 软件系统有限公司开发，广泛应用于自动化控制、机器人科学、电信、数据通信、航空航天、计算机网络系统、医疗仪器设备、交通运输、安全防卫系统、POS 机、零售机等任务关键型应用领域。20 世纪 90 年代后期，QNX 系统在高速增长的因特网终端设备、信息家电及掌上电脑等领域也得到了广泛应用。

QNX 的体系结构决定了它具有非常好的伸缩性，用户可以把应用程序代码和 QNX 内核直接编译在一起，使之为简单的嵌入式应用生成一个单一的多线程映像。它也是世界上第一个遵循 POSIX 1003.1 标准从零设计的微内核，因此具有非常好的可移植性。

华清远见

嵌入式操作系统的选择是前期设计过程的一项重要工作，这将影响到工程后期的发布及软件的维护。不管选用什么样的系统，首先应该考虑操作系统对硬件的支持，如果选择的系统不支持将来要使用的硬件平台，那么这个系统是不合适的；其次要考虑的是开发调试用的工具，特别是对于开销敏感和技术水平不强的企业来说，开发工具往往在开发过程中起决定性作用；最后要考虑的问题是，该系统能否满足应用需求。如果一个操作系统提供出来的 API 很少，那么无论这个系统有多么稳定，应用层也很难进行二次开发，这显然也不是开发人员希望看到的。由此可见，选择一款既能满足应用需求、性价比又可达到最佳的实时操作系统，对开发工作的顺利开展意义非常重大。

1.4 嵌入式Linux基础



随着摩托罗拉手机 A760、IBM 智能型手表 WatchPad、夏普 PDA Zaurus 等一款款高性能“智能数码产品”的出现，以及 Motorola、三星、MontaVista、飞利浦、Nokia、IBM、SUN 等众多国际顶级巨头的加入，嵌入式 Linux 的队伍越来越庞大，在通信、信息、数字家庭、工业控制等领域，随处都能见到嵌入式 Linux 的身影。

究竟是什么原因让嵌入式 Linux 发展如此迅速呢？又究竟是什么原因让它能与强劲的 VxWorks、Window CE 相抗衡呢？这一切还是要归根于它的“父亲”——Linux 的功劳。可以说，嵌入式 Linux 正是继承和发展了 Linux 的诱人之处才走到今天的，而 Linux 也正是有了嵌入式 Linux 的广泛应用才使其更加引人瞩目。下面就从 Linux 开始，一层层揭开嵌入式 Linux 的面纱。

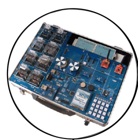
华清远见

1.4.1 Linux发展概述

简单来说，Linux 是指一套免费使用和自由传播的类 UNIX 操作系统。人们通常所说的 Linux 是指 Linus Torvalds 所写的 Linux 操作系统内核。

当时的 Linus 还是芬兰赫尔辛基大学的一名学生，他主修的课程中有一门课是操作系统，而且这门课是专门研究程序的设计和执行。最后这门课程提供了一种称为 Minix 的初期 UNIX 系统。Minix 是一款仅为教学而设计的操作系统，而且功能有限。因此，和 Minix 的众多使用者一样，Linus 也希望能给它添加一些功能。

在之后的几个月里，Linus 根据实际需要，编写了磁盘驱动程序以便下载访问新闻组的文件，又写了个文件系统以便能够阅读 Minix 文件系统中的文件。这样，“当你有了任务切换，有了文件系统和设备驱动程序后，这就是 UNIX，或者至少是其内核”。于是，0.0.1 版本的 Linux 就诞生了。



Linus从一开始就决定自由传播Linux，他把源代码发布到网上，于是众多的爱好者和程序员通过互联网加入到Linux的内核开发工作中。这个思想与FSF（Free Software Foundation）资助发起的GNU（GNU's Not UNIX）的自由软件精神不谋而合。

GNU是为了推广自由软件的精神以实现一个自由的操作系统，然后从应用程序开始，实现其内核。而当时Linux的优良性能备受GNU的赏识，于是GNU决定采用Linus及其开发者的内核。在他们的共同努力下，Linux这个完整的操作系统诞生了。其中的程序开发共同遵守General Public License（GPL）协议，这是最开放也是最严格的许可协议方式，这个协议规定了源代码必须可以无偿地获取并且修改。因此，从严格意义上说，Linux应该称为GNU/Linux，其中许多重要的工具如gcc、gdb、make、Emacs等都是GNU贡献。

这个"婴儿版"的操作系统以平均两星期更新一次的速度迅速成长，如今的Linux已经有超过250种发行版本，并且可以支持所有体系结构的处理器，如X86、PowerPC、ARM、XSCALE等，也可以支持带MMU或不带MMU的处理器。到目前为止，它的内核版本也已经从最开始的0.0.1发展到现在的2.6.xx。

1.4.2　Linux作为嵌入式操作系统的优势

从Linux系统的发展过程可以看出，Linux从最开始就是一个开放的系统，并且始终遵循着源代码开放的原则，它是一个成熟而稳定的网络操作系统。作为嵌入式操作系统，Linux有如下优势。

1．低成本开发系统

Linux的源代码开放性允许任何人获取并修改Linux的源代码。这样，一方面大大降低了开发的成本，另一方面又可以提高开发产品的效率，并且还可以在Linux社区中获得支持。用户只需向邮件列表发一封邮件，即可获得作者的支持。

2．可应用于多种硬件平台

Linux可支持X86、PowerPC、ARM、XSCALE、MIPS、SH、68K、Alpha、SPARC等多种体系结构，并且已经被移植到多种硬件平台。这对于经费、时间受限制的研究与开发项目是很有吸引力的。Linux采用一个统一的框架对硬件进行管理，同时从一个硬件平台到另一个硬件平台的改动与上层应用无关。

3．可定制的内核

Linux具有独特的内核模块机制，它可以根据用户的需要，实时地将某些模块插入到内核中或者从内核中移走，并能根据嵌入式设备的个性需要量体裁衣。经裁减的Linux内核最小可达到150KB以下，尤其适合嵌入式领域中资源受限的情况。当前的2.6内核加入了许多嵌入式友好特性，例如，当设备不需要使用图形界面时，可以屏蔽内核的相应选项。

华清远见

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

4．性能优异

Linux 系统内核精简、高效和稳定，能够充分发挥硬件的功能，因此比其他操作系统的运行效率更高。在个人计算机上使用 Linux，可以将它作为工作站。它也非常适合在嵌入式领域中应用，对比其他操作系统，它占用的资源更少，运行更稳定，速度更快。

5．良好的网络支持

Linux 是首先实现 TCP/IP 协议栈的操作系统，它的内核结构在网络方面是非常完整的，并提供了对包括十兆位、百兆位及千兆位的以太网，还有无线网络、Token Ring（令牌环）和光纤甚至卫星的支持，这对现在依赖于网络的嵌入式设备来说无疑是很好的选择。

1.4.3　Linux 发行版本

由于 Linux 属于 GNU 系统，而这个系统采用的是 GPL 协议，并保证了源代码的公开，于是众多组织或公司在 Linux 内核源代码的基础上进行了一些必要的修改加工，然后开发一些配套的软件，并把它整合成一个自己的发布版 Linux。除去非商业组织 Debian 开发的 Debian GNU/Linux 外，美国的 Red Hat 公司发行了 Red Hat Linux，法国的 Mandrake 公司发行了 Mandrake Linux，德国的 SUSE 公司发行了 SUSE Linux，国内众多公司也发行了中文版的 Linux，如著名的红旗 Linux。Linux 目前已经有超过 250 个发行版本。下面仅对 Red Hat、Debian、Mandrake 等有代表性的 Linux 发行版本进行介绍。

1．Red Hat

目前，国内乃至全世界的 Linux 用户最熟悉的发行版想必就是 Red Hat 了。Red Hat 最早是由 Bob Young 和 Marc Ewing 在 1995 年创建的。目前 Red Hat 分为两个系列：由 Red Hat 公司提供收费技术支持和更新的 Red Hat Enterprise Linux（RHEL，Red Hat 的企业版），以及由社区开发的免费的桌面版 Fedora Core。

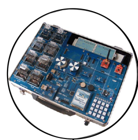
Red Hat 企业版有 3 个版本——AS、ES 和 WS，AS 是其中功能最为强大和完善的版本。而正统的桌面版 Red Hat 版本更新早已停止，最后一版是 Red Hat 9.0。本书就以稳定性高的 RHEL AS 作为安装实例进行讲解。

官方主页：http://www.redhat.com/。

2．Debian

之所以把 Debian 单独列出，是因为 Debian GNU/Linux 是一个非常特殊的版本。1993 年，伊恩·默多克（Ian Murdock）发起 Debian 计划，它的开发模式和 Linux 及其他开放性源代码操作系统的精神一样，都是由超过 800 位志愿者通过互联网合作开发而成的。一直以来，Debian GNU/Linux 被认为是最正宗的 Linux 发行版本，而且它是一个完全免费的、高质量的且与 UNIX 兼容的操作系统。

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com
华清远见



Debian 系统分为 3 个版本，分别为稳定版（Stable）、测试版（Testing）和不稳定版（Unstable）。并且每次发布的版本都是稳定版，而测试版在经过一段时间的测试证明没有问题后会成为新的稳定版。Debian 拥有超过 8 710 种不同的软件，而且每一种软件都是自由的，有非常方便的升级安装指令，基本囊括了用户的需要。Debian 也是最受欢迎的嵌入式 Linux 之一。

官方主页：http://www.debian.org/。

3．国内的发行版本及其他

目前，国内的红旗、新华等都发行了自己的 Linux 版本。

除了前面所提到的这些版本外，业界还存在着诸如 Gentoo、LFS 等适合专业人士使用的版本，在此不做介绍，有兴趣的读者可以自行查找相关资料。

1.4.4 如何学习 Linux

实践出真知，学习 Linux 也一样，只有通过大量的动手实践才能真正领会 Linux 的精髓，才能迅速掌握在 Linux 上的应用开发。因此，在本书中笔者安排了大量的实验环节和课后实践环节，希望读者尽可能多参与。

另外，需要指出的是，互联网也是一个很好的学习工具，一定要充分地加以利用。正如编程语言一样，实践的过程中总会出现多种多样的问题，笔者在写作过程中会尽可能地考虑可能出现的问题，但限于篇幅和实际情况，不可能考虑到所有可能出现的问题，所以希望读者能充分利用互联网这一共享的天空，在其中寻找答案。以下列出了国内的一些 Linux 论坛：

- http://www.linuxfans.org。
- http://www.linuxforum.net/。
- http://www.linuxeden.com/forum/。
- http://www.newsmth.net。

1.5 Linux 安装



有了初步的了解后，读者是否想亲自试一下？其实安装 Linux 是一件很容易的事情，不过在开始安装之前，还需要了解一下在 Linux 安装过程中可能遇到的一些基本知识及其与 Windows 的区别。

1.5.1 基础概念

1．文件系统、分区和挂载

华清远见

华清远见 HQYJ.COM

华清远见教育集团官网：www.hqyj.com

文件系统是指操作系统中与管理文件有关的软件和数据。Linux 的文件系统和 Windows 中的文件系统有很大的区别，Windows 文件系统是以驱动器的盘符为基础的，而且每一个目录与相应的分区对应，例如，“E:\workplace”是指此文件在 E 盘这个分区下；而 Linux 恰好相反，文件系统是一个文件树，并且它的所有文件和外部设备（如硬盘、光驱等）都是以文件的形式挂接在这个文件树上的，如“/usr/local”。总之，在 Windows 操作系统下，目录结构属于分区；在 Linux 操作系统下，分区属于目录结构。其关系如图 1.2 和图 1.3 所示。

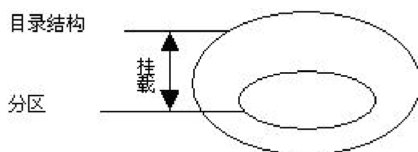


图 1.2　Linux 下目录与分区的关系

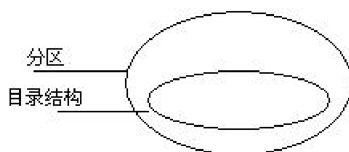


图 1.3　Windows 下目录与分区的关系

因此，在 Linux 中把每一个分区和某一个目录对应，以后对这个目录的操作就是对这个分区的操作，这样就实现了硬件管理手段和软件目录管理手段的统一。这个把分区和目录对应的过程称为**挂载**（Mount），而这个挂载在文件树中的位置就是**挂载点**。这种对应关系可以由用户随时中断和改变。

2．主分区、扩展分区和逻辑分区

硬盘分区是针对一个硬盘进行操作的，它可以分为主分区、扩展分区、逻辑分区。其中，主分区就是包含操作系统启动所必需的文件和数据的硬盘分区。要在硬盘上安装操作系统，则该硬盘必须要有一个主分区，而且其主分区的数量可以是 1～3 个；扩展分区也就是除主分区外的分区，它不能直接使用，必须将其划分为若干个逻辑分区才可使用，其数量可以有 0 或 1 个；而逻辑分区则在数量上没有什么限制。它们的关系如图 1.4 所示。

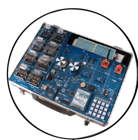


图 1.4　Linux 下主分区、扩展分区、逻辑分区示意图

一般而言，对于先装了 Windows 的用户，Windows 的 C 盘是装在主分区上的，可以把 Linux 安装在另一个主分区或扩展分区上。通常，为了安装方便及安全起见，一般把 Linux 装在多余的逻辑分区上，如图 1.5 所示。

华清远见

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com



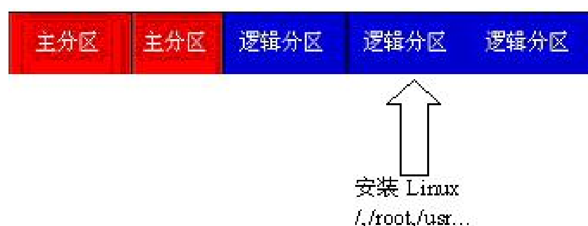


图 1.5　Linux 安装分区示意图

3．SWAP 交换分区

在硬件条件有限的情况下，为了运行大型的程序，Linux 在硬盘上划出一个区域当做临时的内存，Windows 操作系统把这个区域叫做虚拟内存，而 Linux 把它叫做交换分区 SWAP。在安装 Linux 建立交换分区时，一般将其设为内存大小的 2 倍，当然也可以设为更大。

4．分区格式

不同的操作系统选择了不同的格式，同一种操作系统也可能支持多种格式。微软公司的 Windows 就选择了 FAT32、NTFS 两种格式，但是 Windows 不支持 Linux 上常见的分区格式。Linux 是一个开放的操作系统，它最初使用 Ext2 格式，后来使用 Ext3 格式，但是它同时支持非常多的分区格式，包括很多大型机上 UNIX 使用的 XFS 格式，也包括微软公司的 FAT 及 NTFS 格式。

华清远见

5．GRUB

GRUB 是一种引导装入器（类似在嵌入式中非常重要的 Bootloader），它负责装入内核并引导 Linux 系统，位于硬盘的起始部分。由于 GRUB 多方面的优越性，如今的 Linux 一般都默认采用 GRUB 来引导 Linux 操作系统，但事实上它还可以引导 Windows 等多种操作系统。

6．root 权限

Linux 也是一个多用户的系统（这一点类似于 Windows XP），不同的用户和用户组会有不同的权限，其中把具有超级权限的用户称为 root 用户。root 的默认主目录在/root 下，而其他普通用户的目录则在/home 下。root 的权限极高，它甚至可以修改 Linux 的内核，因此建议初学者要慎用 root 权限，否则，一个小小参数的设置错误很有可能导致系统出现严重问题。

1.5.2　硬件需求

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

Linux对硬件的需求非常低。如果只想在字符方式下运行，那么一台386的计算机就可以用来安装Linux；如果想运行X-Window，那么也只需要一台16MB内存、600MB硬盘的486计算机即可。这听起来比那些需要256MB内存、2.0GHz的操作系统要好得多，事实上也正是如此。

现在软件和硬件行业的趋势是让用户购买更快的计算机，不断扩充内存和硬盘，而Linux却不受这个趋势的影响。随着Linux的发展，由于在其上运行的软件越来越多，因此它所需要的配置越来越高，但是用户可以有选择地安装软件，从而节省资源。既可以运行在最新的Pentium 4处理器上，也可以运行在400MHz的Pentium II上，甚至如果用户需要，也可以在只有文本界面的更低配置的机器上运行。由此可见，Linux非常适合需求各异的嵌入式硬件平台，而且Linux可以很好地支持标准配件。如果用户的计算机采用了标准配件，那么运行Linux应该没有任何问题。

1.5.3 安装准备

在开始安装之前，首先需要了解一下机器的硬件配置，包括以下几个问题：

- 有几个硬盘，每个硬盘的大小，如果有两个以上的硬盘哪个是主盘。
- 内存多大。
- 显卡的厂家和型号，有多大的显存。
- 显示器的厂家和型号。
- 鼠标的类型。

如果用户的计算机需要联网，那么还需要注意以下问题：

- 计算机的IP地址，子网掩码，网关，DNS的地址，主机名。
- 有时还需要搞清楚网卡的型号和厂商。

如果不确定系统对硬件的兼容性，或者想了解Linux是否支持一些比较新或不常见的硬件，用户可以到http://hardware.redhat.com和http://xfree86.org进行查询。

其次，用户可以选择从网络安装（如果带宽够大，笔者推荐从商家手中购买Linux的安装盘，一般会获得相应的产品手册、售后服务和众多附赠的商业软件），也可以从他人那里复制。放心，这是合法的，因为Linux是免费的。如果用户需要获得最新的，或需要一个不易于购买到的版本，那么用户可以从http://www.Linuxiso.org下载一个需要的Linux版本。

最后，应在安装前确认磁盘上是否有足够的空间。一般的发行版本全部安装需要3GB左右，最小安装可以到数十兆字节，当然还需要给未来的使用留下足够的空间。如果用户拥有的是一个已经分区的空闲空间，那么可以选择安装前在Windows下删除相应分区，也可以选择在安装时删除。

1.5.4 安装过程

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

Kubuntu 是基于 KDE 的一个非常友好的操作系统，中文名称为“酷班图”，是由 Ubuntu 衍生的一款操作系统，最新版本是 11.04，支持中文；采用 KDE 作为桌面环境，最新版本采用 KDE SC 4.51。作为 Ubuntu 项目的一部分，保持可以预测的 6 个月的发布周期，Kubuntu 对于所有人而言是免费 GNU/Linux 的发行版。Kubuntu 是一个用户界面友好的基于 KDE（K 桌面环境）的操作系统。Kubuntu 和 Ubuntu 的唯一区别就是桌面环境，如果在 Ubuntu 中安装 KDE（并且卸载 GNOME），效果和 Kubuntu 将是一致的。

1. 语言选择

安装向导的第一步询问用户希望安装过程中和最终完成安装后使用的默认语言，如图 1.6 所示。需要注意的是，此处的设置将在稍后的国家和时区设置中直接指向使用该语言的国家和时区。



图 1.6　语言选择

语言选择的操作步骤如下：

（1）在“安装”对话框中间的下拉列表框中查找语言，选择“中文（简体）”选项，“安装”对话框立即显示相应的语言。

（2）单击“安装 Kubuntu”按钮，进入下一步。

2. 准备安装

在正式安装 Kubuntu 之前要确保计算机具备了必要的硬件条件，如至少 3.8GB 的硬盘空间，已插入电源（避免笔记本在安装过程中因电力不足导致的安装失败）并且已接入互联网。以上这些内容 Kubuntu 会自动检测，确认无误后可直接单击“下一步”按钮即可，如图 1.7 所示。

华清远见
HQYJ.COM
华清远见教育集团官网：www.hqyj.com

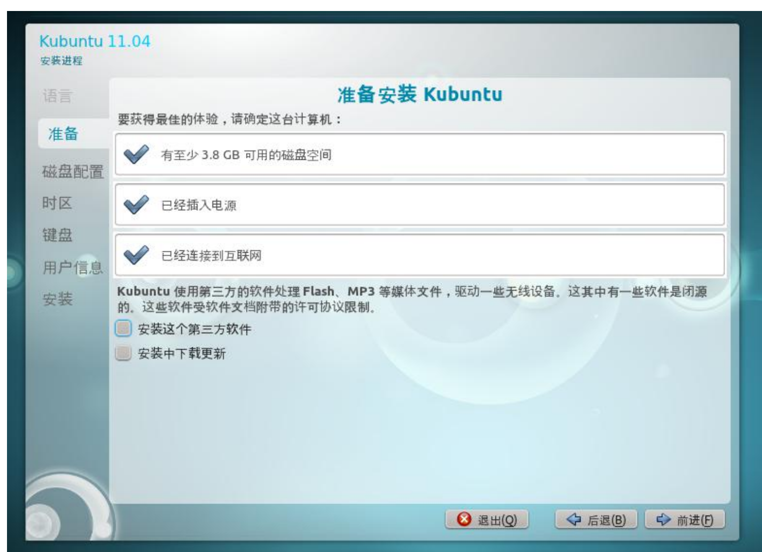


图 1.7　准备安装

3．磁盘配额

为了更深入地学习 Kubuntu 的安装过程，选择手动配置磁盘空间，配置过程分别如图 1.8 到图 1.11 所示。



图 1.8　磁盘配置

华清远见
HQYJ.COM
华清远见教育集团官网：www.hqyj.com



图 1.9　确认选择分区

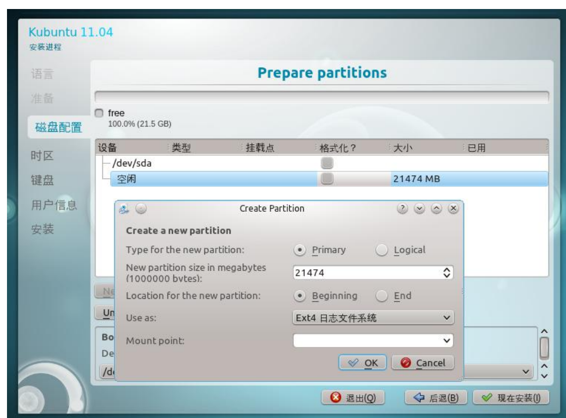


图 1.10　分配磁盘空间

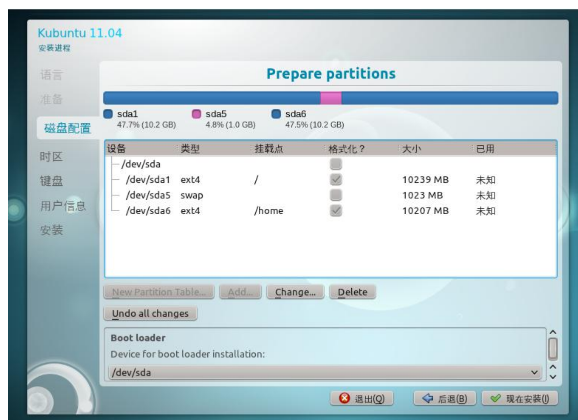


图 1.11　磁盘列表

4. 选择时区

根据第一步用户选择的语言，安装向导在这一步给出候选的国家和城市。如图 1.12 所示，若前一步选择“中文（简体）”，安装向导自动推荐的“地区”为“亚洲”，“时区”为“中国（哈尔滨）”。同时，安装向导也会自动选择相应的时区和当前时间。再次修改“国家/地区”的操作步骤如下：

（1）在“时区”对话框下方的“地区”下拉列表框中选择所在区域。

（2）在“时区”对话框下方的“时区”下拉列表框中选择城市。

（3）单击“前进”按钮，进入下一步。



图 1.12　选择时区

5. 键盘布局

紧接着，安装向导在第四步提示用户选择键盘布局，如图 1.13 所示。由于不同语言使用的键盘也不同，例如，日文键盘是 109 键，而标准键盘通常为 104 键，因此，用户不要随意选择。如果不清楚键盘类型，可以直接选择安装向导推荐的键盘类型。选择键盘布局的操作步骤如下：

（1）在“键盘”对话框的“布局”下拉列表框中选择键盘类型，右侧列表框即可显示该类别的键盘子类即“变种”。

（2）在“键盘”对话框右侧的列表框中选择键盘子类。

（3）确认键盘布局选择是否正确后，单击“前进”按钮，进入下一步。



图 1.13　选择键盘布局

6. 用户信息

安装向导第六步的任务是，要求用户为新系统建立用户账户，并为主机命名，如图 1.14 所示。

第六步的第一个任务是创建用户账户。Linux 操作系统中权限最高的用户是 root，但出于系统安全的考虑，安装程序要求用户创建一个用来取代 root，执行非管理任务的普通用户账号。用户账户信息包括 3 项：用户姓名、账户名、账户密码。安装向导希望输入账户使用者的姓名，建议输入全名；然后为该用户选择账户名，它是在登录时使用的名称；最后，要求输入密码和确认密码。其中账户名要求由数字、小写字母组成，且必须以小写字母开头，不能有空格。如果输入非法账户名，安装向导将警告用户。

第六步的另一项任务是为安装主机命名。该主机名将用于标识主机接入网络后的身份，因此需要用户注意。主机名必须以字母开头，可以包含数字、标点符号，且不能含空格。如果违背命名规则，安装向导同样会给予警告。

第六步的具体操作步骤如下：

（1）分别在“您的姓名”、“选择一个用户名”、“选择一个密码”、“您的计算机名”文本框中输入合法的字符串。

（2）单击“前进”按钮，安装向导将验证用户输入的字符串合法性，如果验证合法，进入下一步。

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com



图 1.14　填写用户信息

7．安装

以上设置完成后 Kubuntu 就进入了正式的安装过程，如图 1.15 到图 1.17 所示。这个过程比较漫长，所以一定要耐心等待，确保中途不要切断电源和网线。

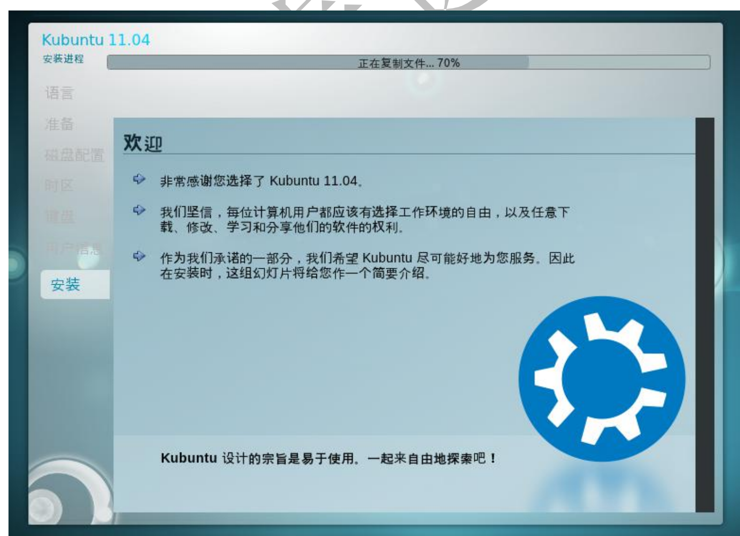


图 1.15　安装过程 1

华清远见
HQYJ.COM
华清远见教育集团官网：www.hqyj.com





图 1.16　安装过程 2

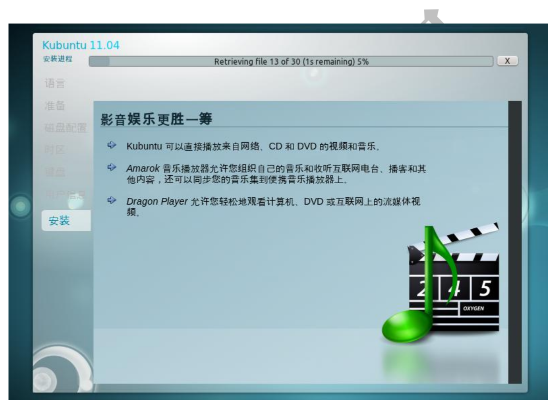


图 1.17　安装过程 3

安装完毕后系统提示重启，单击“现在重启”按钮，如图 1.18 所示。



图 1.18　重启系统

8. 初次登录

第一次安装结束后，系统会自动重新启动，首先打开 Kubuntu 的登录界面，如图 1.19 所示。此时，Kubuntu 的系统安装就完成了。

在登录界面的文本框中输入账户名与口令，按 Enter 键，便进入 Kubuntu 桌面环境，如图 1.20 所示。



图 1.19　登录界面



图 1.20　进入桌面环境

Kubuntu 菜单选项如图 1.21 所示。

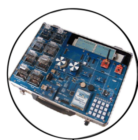


图 1.21　菜单选项

1.6 Linux 文件及文件系统



在安装完 Linux 之后，首先对 Linux 中一些非常重要的概念进行简要介绍，以便进一步学习使用 Linux。



1.6.1 文件类型及文件属性

1. 文件类型

Linux 中的文件类型与 Windows 有显著的区别，其中最显著的区别在于 Linux 对目录和设备都当做文件进行处理，这样就简化了对各种不同类型设备的处理，提高了效率。Linux 中主要的文件类型分为 4 种：普通文件、目录文件、链接文件和设备文件。

1）普通文件

普通文件如同 Windows 中的文件一样，是用户日常使用最多的文件，包括文本文件、Shell 脚本、二进制的可执行程序和各种类型的数据。

2）目录文件

在 Linux 中，目录也是文件，它们包含文件名和子目录名及指向那些文件和子目录的指针。目录文件是 Linux 中存储文件名的唯一地方，当把文件和目录相对应起来时，也就是用指针将其链接起来之后，就构成了目录文件。因此，在对目录文件进行操作时，一般不涉及对文件内容的操作，而只是对目录名和文件名的对应关系进行了操作。

另外，在 Linux 系统中的每个文件都被赋予一个唯一的数值，而这个数值被称作索引节点。索引节点存储在一个称作索引节点表（Inode Table）中，该表在磁盘格式化时被分配。每个实际的磁盘或分区都有其自己的索引节点表。一个索引节点包含文件的所有信息，包括磁盘上数据的地址和文件类型。

Linux 文件系统把索引节点号 1 赋给根目录，这就是 Linux 的根目录文件在磁盘上的地址。根目录文件包括文件名、目录名及它们各自的索引节点号的列表，Linux 可以通过查找从根目录开始的一个目录链来找到系统中的任何文件。

Linux 通过上下链接目录文件系统来实现对整个文件系统的操作。例如，把文件从一个磁盘目录移到另一个磁盘的目录时（实际上是通过读取索引节点表来检测这种行动），这时，原先文件的磁盘索引号删除，而在新磁盘上建立相应的索引节点。它们之间的相应关系如图 1.22 所示。

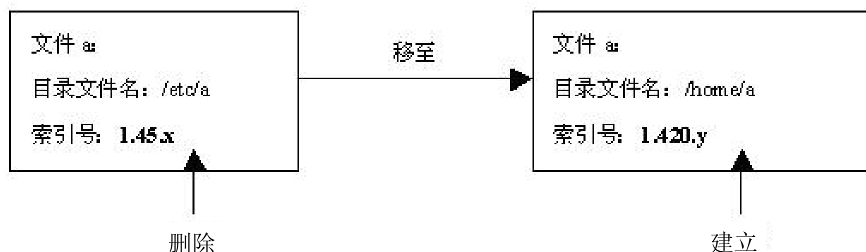


图 1.22 目录文件与索引节点的关系

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

3）链接文件

链接文件类似于 Windows 中的“快捷方式”，但是它的功能更为强大。它可以实现对不同的目录、文件系统甚至是不同的机器上的文件直接访问，并且不必重新占用磁盘空间。

4）设备文件

Linux 把设备都当做文件一样来进行操作，这样就大大方便了用户的使用（在后面的 Linux 编程中可以更为明显地看出）。在 Linux 下，与设备相关的文件一般都在/dev 目录下，它包括两种，一种是块设备文件，另一种是字符设备文件。

- 块设备文件是指数据的读/写设备，它们是以块（如由柱面和扇区编址的块）为单位的设备，最简单的如硬盘（/dev/hda1）等。
- 字符设备主要是指串行端口的接口设备。

2．文件属性

Linux 中的文件属性如图 1.23 所示。

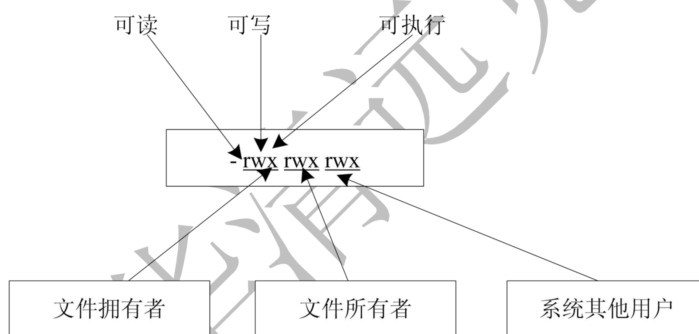


图 1.23　Linux 文件属性表示方法

首先，Linux 中文件的拥有者可以把文件的访问属性设成 3 种不同的访问权限：可读（r）、可写（w）和可执行（x）。文件又有 3 个不同的用户级别：文件拥有者（u）、所属的用户组（g）和系统中的其他用户（o）。

第一个字符显示文件的类型如下：

- “-”表示普通文件。
- “d”表示目录文件。
- “l”表示链接文件。
- “c”表示字符设备。
- “b”表示块设备。
- “p”表示命名管道，如 FIFO（First In First Out，先进先出）文件。

- “f”表示堆栈文件，如 LIFO（Last In First Out，后进先出）文件。

第一个字符之后有 3 个三位字符组：

- 第一个三位字符组表示文件拥有者（u）对该文件的权限。
- 第二个三位字符组表示文件用户组（g）对该文件的权限。
- 第三个三位字符组表示系统其他用户（o）对该文件的权限。
- 若该用户组对此没有权限，一般显示“-”字符。

1.6.2 文件系统类型介绍

1．Ext2 和 Ext3

Ext3 是现在 Linux（包括 Red Hat，Mandrake 下）常见的默认文件系统，它是 Ext2 的升级版本。正如 Red Hat 公司的首席核心开发人员 Michael K.Johnson 所说，从 Ext2 转换到 Ext3 主要有以下 4 个理由：可用性、数据完整性、速度及易于转化。Ext3 中采用了日志式的管理机制，它使文件系统具有很强的快速恢复能力，并且由于从 Ext2 转换到 Ext3 无须进行格式化，因此，更加推进了 Ext3 文件系统的推广。

2．SWAP 文件系统

该文件系统是 Linux 中作为交换分区使用的。在安装 Linux 时，交换分区是必须建立的，并且它所采用的文件系统类型必须是 SWAP 而没有其他选择。

3．VFAT 文件系统

Linux 中，把 DOS 中采用的 FAT 文件系统（包括 FAT12、FAT16 和 FAT32）都称为 VFAT 文件系统。

4．NFS 文件系统

NFS 文件系统是指网络文件系统，这种文件系统也是 Linux 的独到之处。它可以很方便地在局域网内实现文件共享，并且使多台主机共享同一主机上的文件系统。而且 NFS 文件系统访问速度快、稳定性高，已经得到了广泛的应用，尤其是在嵌入式领域，使用 NFS 文件系统可以很方便地实现文件本地修改，从而免去了一次次读/写 Flash 的忧虑。

5．ISO 9660 文件系统

这是光盘所使用的文件系统，在 Linux 中对光盘已有了很好的支持，它不仅可以提供对光盘的读/写，还可以实现对光盘的刻录。

1.6.3 Linux 目录结构

Linux 的目录结构如图 1.24 所示。下面以 Red Hat Enterprise 4 AS 为例，详细列出了 Linux 文件系统中各主要目录的存放内容，如表 1.1 所示。

华清远见

表 1.1　Linux 文件系统目录结构

目　　录	目 录 内 容
/bin	bin 就是二进制（binary）的英文缩写。在这里存放前面 Linux 常用操作命令的执行文件，如 mv、ls、mkdir 等。有时，该目录的内容和/usr/bin 里面的内容一样，它们都是放置一般用户使用的执行文件
/boot	该目录下存放操作系统启动时所要用到的程序。如启动 grub 就会用到其下的/boot/grub 子目录
/dev	该目录中包含了所有 Linux 系统中使用的外部设备。要注意的是，这里并不是存放的外部设备的驱动程序，它实际上是一个访问这些外部设备的端口。由于在 Linux 中所有的设备都当做文件进行操作，比如/dev/cdrom 代表光驱，用户可以非常方便地像访问文件、目录一样对其进行访问
/etc	该目录下存放了系统管理时要用到的各种配置文件和子目录，如网络配置文件、文件系统、x 系统配置文件、设备配置信息、设置用户信息等都在该目录下。系统在启动过程中需要读取其参数进行相应的配置
/etc/rc.d	该目录主要存放 Linux 启动和关闭时要用到的脚本文件，在后面的启动详解中还会进一步讲解
/etc/rc.d/init	该目录存放所有 Linux 服务默认的启动脚本（在新版本的 Linux 中还用到的是/etc/xinetd.d 目录下的内容）
/home	该目录是 Linux 系统中默认的用户工作根目录。在 2.1.1 节中将讲到，执行 adduser 命令后系统会在/home 目录下为对应账号建立一个同名的主目录
/lib	该目录用来存放系统动态链接共享库，几乎所有的应用程序都会用到这个目录下的共享库。因此，千万不要轻易对这个目录进行什么操作
/lost+found	该目录在大多数情况下都是空的。只有当系统产生异常时，会将一些遗失的片段放在此目录下
/media	该目录下是光驱和软驱的挂载点，Fedora Core 4 已经可以自动挂载光驱和软驱
/misc	该目录下存放从 DOS 下进行安装的实用工具，一般为空
/mnt	该目录是软驱、光驱、硬盘的挂载点，也可以临时将别的文件系统挂载到此目录下
/proc	该目录用于放置系统核心与执行程序所需的一些信息，而这些信息是在内存中由系统产生的，故不占用硬盘空间
/root	该目录是超级用户登录时的主目录
/sbin	该目录用来存放系统管理员常用的系统管理程序
/tmp	该目录用来存放不同程序执行时产生的临时文件，也作为 Linux 安装软件的默认安装路径
/usr	这是一个非常重要的目录，用户的很多应用程序和文件都存放在这个目录下，类似于 Windows 下的 Program Files 目录
/usr/bin	该目录用来存放系统用户使用的应用程序
/usr/sbin	该目录用来存放超级用户使用的比较高级的管理程序和系统守护程序
/usr/src	内核源代码默认的放置目录
/srv	该目录存放一些服务启动之后需要提取的数据
/sys	这是 Linux 2.6 内核的一个很大的变化，该目录下安装了 2.6 内核中新出现的一个文件系统 sysfs sysfs 文件系统集成了 3 种文件系统的信息：针对进程信息的 proc 文件系统、针对设备的 devfs 文件系统，以及针对伪终端的 devpts 文件系统。该文件系统是内核设备树的一个直观反映。当一个内核对象被创建的时候，对应的文件和目录也在内核对象子系统中被创建
/var	这也是一个非常重要的目录，很多服务的日志信息都存放在这里

华清远见

华清远见 HQYJ.COM

华清远见教育集团官网：www.hqyj.com

```
/
├── /bin
├── /boot ── grub
├── /dev
├── /etc ── rc.d ── cron.d
├── /home
├── /lib
├── /mnt
├── /medea ── cdrom ── floppy
├── /misc
├── /net
├── /opt
├── /proc ── net ── sys ── tty
├── /sbin
├── /srv
├── /sys ── block ── bus ── devicess
├── /usr ── bin ── sbin ── src
├── /tmp
└── /var
```

图 1.24　Linux 目录结构

华清远见

1.7 本章习题



1. 什么是操作系统？
2. 说出你知道的几种操作系统。
3. 练习安装 Linux，并明确分区的规划。
4. 什么是交换分区？
5. 在 Linux 系统中，存放二进制命令文件的目录有哪几个？

华清远见 HQYJ.COM

华清远见

第 2 章 Linux 操作系统使用与系统配置

Linux 是一个高可靠、高性能的系统，而所有这些优越性只有在直接使用 Linux 命令行时（Shell 环境）才能充分地体现出来。通过本章的学习，让读者能够掌握一些基本的 Linux 命令，并能独立定制 Linux 中的系统服务。

华清远见 HQYJ.COM

华清远见教育集团官网：www.hqyj.com

2.1 Linux 基本命令



在安装完 Linux 再次启动之后，就可以进入到与 Windows 类似的图形化界面了，这个界面就是 Linux 图形化界面 X 窗口系统（简称 X）的一部分。要注意的是，X 窗口系统仅仅是 Linux 上的一个软件（或者也可称为服务），它不是 Linux 自身的一部分。虽然现在的 X 窗口系统已经与 Linux 整合得相当好了，但毕竟还不能保证绝对的可靠性。另外，X 窗口系统是一个相当耗费系统资源的软件，它会大大降低 Linux 的系统性能。因此，若是希望更好地享受 Linux 所带来的高效及高稳定性，建议读者尽可能地使用 Linux 的命令行界面，也就是 Shell 环境。

当用户在命令行下工作时，不是直接同操作系统内核交互信息，而是由命令解释器接受命令，分析后再传给相关的程序。Shell 是一种 Linux 中的命令行解释程序，就如同 Command.com 是 DOS 下的命令解释程序一样，为用户提供使用操作系统的接口。它们之间的关系如图 2.1 所示。用户在提示符下输入的命令都由 Shell 先解释，然后传给 Linux 内核。

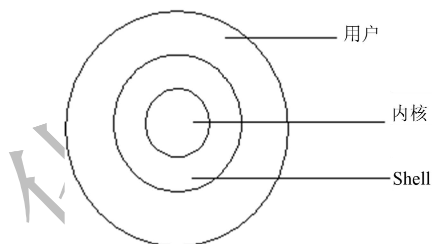


图 2.1　内核、Shell 和用户的关系

Linux 中运行 Shell 的环境是“系统工具”下的“终端”，读者可以单击“终端”以启动 Shell 环境。这时屏幕上显示类似“[linux@www home]$”的信息，其中，linux 是指系统用户，home 是指当前所在的目录。

由于 Linux 中的命令非常多，要全部介绍是不可能的。因此，在本书中按照命令的用途进行分类讲解，并且对每一类中最常用的命令进行详细讲解，同时列出同一类中的其他命令。由于同一类的命令都有很大的相似性，因此，读者通过学习本书中所列命令，可以很快地掌握其他命令。

命令格式说明：

- 格式中带[]的表明为可选项，其他为必选项。
- 选项可以多个连带写入。

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

- 本章后面选项参数列表中加粗的含义是：该选项是非常常用的选项。

2.1.1 用户系统相关命令

Linux 是一个多用户的操作系统，每个用户又可以属于不同的用户组，下面首先来熟悉一下 Linux 中的用户切换和用户管理的相关命令。

1. 用户切换（su）

1）作用

变更为其他使用者的身份，主要用于将普通用户身份转变为超级用户，而且需要输入相应的用户密码。

2）格式

su [选项] [使用者]

其中，使用者为要变更的对应使用者。

3）常见参数

主要选项参数如表 2.1 所示。

表 2.1 su 命令常见参数列表

选　　项	参 数 含 义
-，-l，--login	为该使用者重新登录，大部分环境变量（如 HOME、SHELL 和 USER 等）和工作目录都是以该使用者（USER）为主。若没有指定 USER，默认情况是 root
-m，-p	执行 su 时不改变环境变量
-c，--command	变更账号为 USER 的使用者，并执行指令（command）后再变回原来的使用者

4）使用示例

```
[linuxlinux@www linux]$ su - root
Password:
[root@www root]#
```

示例通过 su 命令将普通用户变更为 root 用户，并使用选项“-”携带 root 环境变量。

5）使用说明

（1）在将普通用户变更为 root 用户时建议使用“-”选项，这样可以将 root 的环境变量和工作目录同时带入，否则，在以后的使用中可能会由于环境变量的原因而出错。

（2）在转变为 root 权限后，提示符变为#。

2. 用户管理（useradd 和 passwd）

Linux 中常见用户管理命令如表 2.2 所示，本书仅以 useradd 和 passwd 为例进行详细讲解，其他命令类似，请读者自行学习使用。

华清远见

表 2.2　Linux 常见用户管理命令

命　令	命令含义	格　式
useradd	添加用户账号	useradd [选项] 用户名
usermod	设置用户账号属性	usermod [选项] 属性值
userdel	删除对应用户账号	userdel [选项] 用户名
groupadd	添加组账号	groupadd [选项] 组账号
groupmod	设置组账号属性	groupmod [选项] 属性值
groupdel	删除对应组账号	groupdel [选项] 组账号
passwd	设置账号密码	passwd [对应账号]
id	显示用户 ID、组 ID 和用户所属的组列表	id [用户名]
groups	显示用户所属的组	groups [组账号]
who	显示登录到系统的所有用户	who

1）作用

（1）useradd：添加用户账号。

（2）passwd：更改对应用户账号密码。

2）格式

（1）useradd：

```
useradd [选项] 用户名
```

（2）passwd：

```
passwd [选项] [用户名]
```

其中，用户名为修改账号密码的用户，若不带用户名，默认为更改当前使用者账号密码。

3）常用参数

（1）useradd 主要选项参数如表 2.3 所示。

表 2.3　useradd 命令常见参数列表

选　项	参数含义
-g	指定用户所属的群组
-m	自动建立用户的登入目录
-n	取消建立以用户名称为名的群组

（2）passwd：一般很少使用选项参数。

4）使用实例

```
[root@www root]# useradd ycw
[root@www root]# passwd ycw
New password:
```


华清远见 HQYJ.COM　华清远见教育集团官网：www.hqyj.com

```
Retype new password:
passwd: all authentication tokens updated successfully
[root@www root]# su - ycw
[ycwycw@www ycw]$
[ycw@www ycw]$ pwd（查看当前目录）
/home/ycw
```

实例中先添加了用户名为 ycw 的用户，接着又为该用户设置了账号密码。并且从 su 命令可以看出，该用户添加成功，其工作目录为/home/ycw。

5）使用说明

（1）在添加用户时，这两个命令是一起使用的，其中，useradd 必须用 root 的权限。而且 useradd 指令所建立的账号实际上是保存在/etc/passwd 文本文件中，文件中每一行包含一个账号信息。

（2）在默认情况下，useradd 所做的初始化操作包括在/home 目录下为对应账号建立一个同名的主目录，并且还为该用户单独建立一个与用户名同名的组。

（3）adduser 只是 useradd 的符号链接（关于符号链接的概念在本节后面会有介绍），两者是相同的。

（4）passwd 还可用于普通用户修改账号密码。Linux 并不采用类似 Windows 的密码回显（显示为*号），所以输入的这些字符用户是看不到的。密码最好包括字母、数字和特殊符号，并且设成 6 位以上。

3. 系统管理命令（ps 和 kill）

Linux 中常见的系统管理命令如表 2.4 所示，本书以 ps 和 kill 为例进行讲解。

表 2.4　Linux 常见系统管理命令

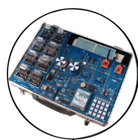
命　　令	命 令 含 义	格　　式
ps	显示当前系统中由该用户运行的进程列表	ps [选项]
top	动态显示系统中运行的程序（一般为每隔 5s）	top
kill	输出特定的信号给指定 PID（进程号）的进程	kill [选项] 进程号（PID）
uname	显示系统的信息（可加选项-a）	uname [选项]
setup	系统图形化界面配置	setup
crontab	循环执行例行性命令	crontab [选项]
shutdown	关闭或重启 Linux 系统	shutdown [选项] [时间]
uptime	显示系统已经运行了多长时间	uptime
clear	清除屏幕上的信息	clear

1）作用

（1）ps：显示当前系统中由该用户运行的进程列表。

（2）kill：输出特定的信号给指定 PID（进程号）的进程，并根据该信号完成指定的行为。其中可能的信号有进程挂起、进程等待、进程终止等。

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com



2）格式

（1）ps：

```
ps [选项]
```

（2）kill：

```
kill [选项] 进程号（PID）
```

kill 命令中的进程号为信号输出的指定进程的进程号，当选项是默认时为输出终止信号给该进程。

3）常见参数

（1）ps 主要选项参数如表 2.5 所示。

表 2.5　ps 命令常见参数列表

选　　项	参 数 含 义
-ef	查看所有进程及其 PID（进程号）、系统时间、命令详细目录、执行者等
-aux	除可显示-ef 所有内容外，还可显示 CPU 及内存占用率、进程状态
-w	显示加宽并且可以显示较多的信息

（2）kill 主要选项参数如表 2.6 所示。

表 2.6　kill 命令常见参数列表

选　　项	参 数 含 义
-s	根据指定信号发送给进程
-p	打印出进程号（PID），但并不送出信号
-l	列出所有可用的信号名称

4）使用实例

```
[root@www root]# ps -ef
UID        PID  PPID  C STIME TTY         TIME CMD
root         1     0  0  2005 ?       00:00:05 init
root         2     1  0  2005 ?       00:00:00 [keventd]
root         3     0  0  2005 ?       00:00:00 [ksoftirqd_CPU0]
root         4     0  0  2005 ?       00:00:00 [ksoftirqd_CPU1]
root      7421     1  0  2005 ?       00:00:00 /usr/local/bin/ntpd -c /etc/ntp.
root     21787 21739  0 17:16 pts/1     00:00:00 grep ntp
[root@www root]# kill 7421
[root@www root]# ps -ef|grep ntp
root     21789 21739  0 17:16 pts/1    00:00:00 grep ntp
```

该实例中首先查看所有进程，并终止进程号为 7421 的 ntp 进程，之后再次查看时已经没有该进程号的进程。

5）使用说明

（1）ps 在使用中通常可以与其他一些命令结合起来使用，主要作用是提高效率。

华清远见

（2）ps 选项中的参数 w 可以写多次，通常最多写 3 次，它的含义表示加宽 3 次，这足以显示很长的命令行了。例如，ps –auxwww。

4．磁盘相关命令（fdisk）

Linux 中与磁盘相关的命令如表 2.7 所示，本书仅以 fdisk 为例进行讲解。

表 2.7　Linux 常见磁盘相关命令

选　项	参 数 含 义	格　　式
free	查看当前系统内存的使用情况	free [选项]
df	查看文件系统的磁盘空间占用情况	df [选项]
du	统计目录（或文件）所占磁盘空间的大小	du [选项]
fdisk	查看硬盘分区情况及对硬盘进行分区管理	fdisk [-l]

1）作用

fdisk 可以查看硬盘分区情况，并可对硬盘进行分区管理，这里主要向读者介绍查看硬盘分区情况。另外，fdisk 也是一个非常好的硬盘分区工具，感兴趣的读者可以另外查找资料学习使用 fdisk 进行硬盘分区。

2）格式

```
fdisk [-l]
```

3）使用实例

```
[root@linux ~]# fdisk -l
Disk /dev/hda: 40.0 GB, 40007761920 bytes
240 heads, 63 sectors/track, 5168 cylinders
Units = cylinders of 15120 * 512 = 7741440 bytes
   Device Boot      Start         End      Blocks   Id  System
/dev/hda1   *           1        1084     8195008+   c  W95 FAT32 (LBA)
/dev/hda2            1085        5167    30867480    f  W95 Ext'd (LBA)
/dev/hda5            1085        2439    10243768+   b  W95 FAT32
/dev/hda6            2440        4064    12284968+   b  W95 FAT32
/dev/hda7            4065        5096     7799526   83  Linux
/dev/hda8            5096        5165      522081   82  Linux swap
```

可以看出，使用“fdisk –l”列出了文件系统的分区情况。

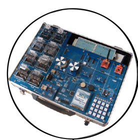
4）使用说明

（1）使用 fdisk 必须拥有 root 权限。

（2）IDE 硬盘对应的设备名称分别为 hda、hdb、hdc 和 hdd，SCSI 硬盘对应的设备名称则为 sda、sdb……此外，hda1 代表 hda 的第一个硬盘分区，hda2 代表 hda 的第二个分区，依此类推。

（3）通过查看/var/log/messages 文件，可以找到 Linux 系统已辨认出来的设备代号。

华清远见 HQYJ.COM　华清远见教育集团官网：www.hqyj.com



5．磁盘挂载命令（mount）

1）作用

挂载文件系统，它的使用权限是超级用户或/etc/fstab 中允许的使用者。正如 1.5.1 节中所述，挂载是指把分区和目录对应的过程，而挂载点是指挂载在文件树中的位置。mount 命令可以把文件系统挂载到相应的目录下，并且由于 Linux 中把设备都当做文件一样使用，因此，mount 命令也可以挂载不同的设备。

通常，在 Linux 下/mnt 目录是专门用于挂载不同的文件系统的，它可以在该目录下新建不同的子目录来挂载不同的设备文件系统。

2）格式

```
mount [选项] [类型] 设备文件名 挂载点目录
```

其中，“类型”是指设备文件的类型。

3）常见参数

mount 命令常见参数如表 2.8 所示。

表 2.8　mount 命令选项常见参数列表

选　项	参 数 含 义
-a	依照/etc/fstab 的内容装载所有相关的硬盘
-l	列出当前已挂载的设备、文件系统的名称和挂载点
-t	将后面的设备以指定类型的文件格式装载到挂载点上。常见的类型有 VFAT、Ext3、Ext2、ISO 9660、NFS 等
-f	通常用于除错。它会使 mount 不执行实际挂上的动作，而是模拟整个挂上的过程，通常会和-v 一起使用

4）使用实例

使用 mount 命令主要通过以下几个步骤：

（1）确认是否为 Linux 可以识别的文件系统。Linux 可识别的文件系统包括以下几种。

- Windows 95/98 常用的 FAT32 文件系统：VFAT。
- WinNT/2000 的文件系统：NTFS。
- OS/2 用的文件系统：HPFS。
- Linux 用的文件系统：Ext2、Ext3、NFS。
- CD-ROM 光盘用的文件系统：ISO 9660。

（2）确定设备的名称，可通过使用命令“fdisk -l”查看。

（3）查找挂载点。

必须确定挂载点已经存在，也就是在/mnt 下的相应子目录已经存在。一般，建议在/mnt 下新建几个如/mnt/windows、/mnt/usb 的子目录，现在有些新版本的 Linux（如红旗 Linux、中软 Linux、MandrakeLinux）都可自动挂载文件系统，Red Hat 仅可自动挂载光驱。

华清远见

（4）挂载文件系统如下所示：

```
[root@linux mnt]# mount -t vfat /dev/hda1 /mnt/c
[root@linux mnt]# cd /mnt/c
24.s03e01.pdtv.xvid-sfm.rmvb  Documents and Settings  Program Files
24.s03e02.pdtv.xvid-sfm.rmvb  Downloads                Recycled
…
```

C 盘是笔者 Windows 系统的启动盘。可见，在挂载了 C 盘之后，可直接访问 Windows 下 C 盘的内容。

（5）在使用完该设备文件后可使用 umount 命令将其卸载。

```
[root@linux mnt]# umount /mnt/c
[root@linux mnt]# cd /mnt/c
[root@linux c]# ls
```

可见，此时目录/mnt/c 下为空。Windows 下的 C 盘成功卸载。

2.1.2 文件目录相关命令

由于 Linux 中有关文件目录的操作非常重要，也非常常用，因此在本节中，将讲解所有的文件操作命令。

1. cd

1）作用

改变工作目录。

2）格式

```
cd [路径]
```

其中，路径为要改变的工作目录，可为相对路径或绝对路径。

3）使用实例

```
[root@www uclinux]# cd /home/linux/
[root@www linux]# pwd
[root@www linux]# /home/linux/
```

该实例中变更工作目录为/home/linux/，在后面的 pwd（显示当前目录）的结果中可以看出。

4）使用说明

（1）该命令将当前目录改变至指定路径的目录。若没有指定路径，则回到用户的主目录。为了改变到指定目录，用户必须拥有对指定目录的执行和读权限。

（2）该命令可以使用通配符。

（3）可使用“cd –”回到前次工作目录。

（4）“./”代表当前目录，“../”代表上级目录。

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

2．ls

1）作用

列出目录的内容。

2）格式

```
ls [选项] [文件]
```

其中，文件选项为查看指定文件的相关内容，若未指定文件，默认查看当前目录下的所有文件。

3）常见参数

ls 命令主要选项参数如表 2.9 所示。

表 2.9　ls 命令常见参数列表

选　　项	参 数 含 义
-1，--format=single-column	一行输出一个文件（单列输出）
-a，-all	列出目录中所有文件，包括以“.”开头的文件
-d	将目录名和其他文件一样列出，而不是列出目录的内容
-l，--format=long，--format=verbose	除每个文件名外，增加显示文件类型、权限、硬链接数、所有者名、组名、大小（Byte）及时间信息（如未指明是其他时间即指修改时间）
-f	不排序目录内容，按它们在磁盘上存储的顺序列出

4）使用实例

```
[ycwing@www /]$ ls -l
total 220
drwxr-xr-x    2 root     root         4096 Mar 31  2005 bin
drwxr-xr-x    3 root     root         4096 Apr  3  2005 boot
-rw-r--r--    1 root     root            0 Apr 24  2002 test.run
…
```

该实例查看当前目录下的所有文件，并通过选项-l 显示出详细信息。

显示格式说明如下：

```
文件类型与权限  链接数  文件属主  文件属组  文件大小  修改的时间  名字
```

5）使用说明

（1）在 ls 的常见参数中，-l（长文件名显示格式）选项是最为常见的，使用它可以详细显示出各种信息。

（2）若想显示出所有以“.”开头的文件，可以使用-a，这在嵌入式的开发中很常用。

3．mkdir

1）作用

创建一个目录。

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

2）格式

```
mkdir [选项] 路径
```

3）常见参数

mkdir 命令主要选项参数如表 2.10 所示

表 2.10　mkdir 命令常见参数列表

选　　项	参 数 含 义
-m	对新建目录设置存取权限，也可以用 chmod 命令（在本节后会有详细说明）设置
-p	可以是一个路径名称。此时若此路径中的某些目录尚不存在，在加上此选项后，系统将自动建立好那些尚不存在的目录，即一次可以建立多个目录

4）使用实例

```
[root@www linux]# mkdir -p ./hello/my
[root@www my]# pwd（查看当前目录命令）
/home/linux/hello/my
```

该实例使用选项“-p”一次创建了./hello/my 多级目录。

```
[root@www my]# mkdir -m 777 ./why
[root@www my]# ls -l
total 4
drwxrwxrwx    2 root     root         4096 Jan 14 09:24 why
```

该实例使用选项“-m”创建了相应权限的目录。对于“777”的权限在本节后面会有详细的说明。

5）使用说明

该命令要求创建目录的用户在创建路径的上级目录中具有写权限，并且路径名不能是当前目录中已有的目录或文件名称。

4．cat

1）作用

连接并显示指定的一个和多个文件的有关信息。

2）格式

```
cat[选项]文件 1 文件 2…
```

其中，文件 1、文件 2 为要显示的多个文件。

3）常见参数

cat 命令常见参数如表 2.11 所示。

表 2.11　cat 命令常见参数列表

选　　项	参 数 含 义

华清远见 HQYJ.COM 华清远见教育集团官网：www.hqyj.com

-n	由第一行开始对所有输出的行数编号
-b	和-n 相似，只不过对于空白行不编号

4）使用实例

```
[ycw@www ycw]$ cat -n hello1.c hello2.c
     1  #include <stdio.h>
     2  void main()
     3  {
     4          printf("Hello!This is my home!\n");
     5  }
     6  #include <stdio.h>
     7  void main()
     8  {
     9          printf("Hello!This is your home!\n");
    10  }
```

在该实例中，指定对 hello1.c 和 hello2.c 进行输出，并指定行号。

5．cp、mv 和 rm

1）作用

（1）cp：将给出的文件或目录复制到另一个文件或目录中。

（2）mv：为文件或目录改名或将文件由一个目录移入另一个目录中。

（3）rm：删除一个目录中的一个或多个文件或目录。

2）格式

（1）cp：

```
cp [选项] 源文件或目录 目标文件或目录
```

（2）mv：

```
mv [选项] 源文件或目录 目标文件或目录
```

（3）rm：

```
rm [选项] 文件或目录
```

3）常见参数

（1）cp 命令主要选项参数如表 2.12 所示。

表 2.12 cp 命令常见参数列表

选　　项	参 数 含 义
-a	保留链接、文件属性，并复制其子目录，其作用等于 dpr 选项的组合
-d	复制时保留链接
-f	删除已经存在的目标文件而不提示
-i	在覆盖目标文件之前将给出提示要求用户确认，回答 y 时目标文件将被覆盖，而且是交互式复制
-p	此时 cp 除复制源文件的内容外，还将把其修改时间和访问权限也复制到新文件中

-r	若给出的源文件是一个目录文件，cp 将递归复制该目录下所有的子目录和文件，此时目标文件必须是一个目录名

（2）mv 命令主要选项参数如表 2.13 所示。

表 2.13　mv 命令常见参数列表

选　　项	参 数 含 义
-i	若 mv 操作将导致对已存在的目标文件的覆盖，此时系统询问是否重写，并要求用户回答 y 或 n，这样可以避免误覆盖文件
-f	禁止交互操作。在 mv 操作要覆盖某已有的目标文件时不给任何指示，在指定此选项后，i 选项将不再起作用

（3）rm 命令主要选项参数如表 2.14 所示。

表 2.14　rm 命令常见参数列表

选　　项	参 数 含 义
-i	进行交互式删除
-f	忽略不存在的文件，但从不给出提示
-r	指示 rm 将参数中列出的全部目录和子目录均递归地删除

4）使用实例

（1）cp 使用实例如下：

```
[root@www hello]# cp -a ./my/why/ ./
[root@www hello]# ls
my  why
```

该实例使用-a 选项将/my/why 目录下的所有文件复制到当前目录下，而此时在原目录下还有原有的文件。

（2）mv 使用实例如下：

```
[root@www hello]# mv -i ./my/why/ ./
[root@www hello]# ls
my  why
```

该实例中把/my/why 目录下的所有文件移至当前目录，则原目录下文件被自动删除。

（3）rm 使用实例如下：

```
[root@www hello]# rm -r -i ./why
rm: descend into directory './why'? y
rm: remove './why/my.c'? y
rm: remove directory './why'? y
```

该实例使用“-r”选项删除“./why”目录下的所有内容，系统会进行确认是否删除。

5）使用说明

（1）cp：该命令把指定的源文件复制到目标文件或把多个源文件复制到目标目录中。

（2）mv：

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

① 该命令根据命令中第二个参数类型的不同（是目标文件还是目标目录）来判断是重命名还是移动文件。当第二个参数类型是文件时，mv 命令完成文件重命名，此时，它将所给的源文件或目录重命名为给定的目标文件名。

② 当第二个参数是已存在的目录名称时，mv 命令将各参数指定的源文件均移至目标目录中。

③ 在跨文件系统移动文件时，mv 先复制，再将原有文件删除，而链至该文件的链接也将丢失。

（3）rm：

① 如果没有使用- r 选项，则 rm 不会删除目录。

② 使用该命令时一旦文件被删除，它是不能被恢复的，所以最好使用-i 参数。

6. chown 和 chgrp

1）作用

（1）chown：修改文件所有者和组别。

（2）chgrp：改变文件的组所有权。

2）格式

（1）chown：

```
chown [选项]...文件所有者[所有者组名] 文件
```

其中，文件所有者为修改后的文件所有者。

（2）chgrp：

```
chgrp [选项]... 文件所有组 文件
```

其中，文件所有组为改变后的文件组拥有者。

3）常见参数

chown 和 chgrp 的常见参数意义相同，其主要选项参数如表 2.15 所示。

表 2.15 chown 和 chgrp 命令常见参数列表

选　　项	参 数 含 义
-c，-changes	详尽地描述每个 file 实际改变了哪些所有权
-f，--silent，--quiet	不打印文件所有权就不能修改的报错信息

4）使用实例

在笔者的系统中一个文件的所有者原来是这样的：

```
[root@www linux]# ls -l
-rwxr-xr-x   15 apectel  linux        4096  6月  4  2005 uClinux-dist.tar
```

华清远见

可以看出，这是一个文件，它的文件拥有者是 apectel，具有可读写和执行的权限，它所属的用户组是 linux，具有可读和执行的权限，但没有可写的全权。同样，系统其他用户对其也只有可读和执行的权限。

首先使用 chown 命令将文件所有者改为 root。

```
[root@www linux]# chown root uClinux-dist.tar
[root@www linux]# ls -l
-rwxr-xr-x   15 root     linux        4096  6月  4  2005 uClinux-dist.tar
```

可以看出，此时，该文件拥有者变为了 root，它所属文件用户组不变。

接着使用 chgrp 命令将文件用户组变为 root。

```
[root@www linux]# chgrp root uClinux-dist.tar
[root@www linux]# ls -l
-rwxr-xr-x   15 root     root        4096  6月  4  2005 uClinux-dist.tar
```

5）使用说明

使用 chown 和 chgrp 命令必须拥有 root 权限。

7. chmod

1）作用

改变文件的访问权限。

2）格式

chmod 可使用符号标记进行更改和八进制数指定更改两种方式，因此它的格式也有两种不同的形式。

（1）符号标记：

```
chmod [选项]…符号权限[符号权限]…文件
```

其中，符号权限可以指定为多个，也就是说，可以指定多个用户级别的权限，但它们中间要用逗号分开表示，若没有显式指出则表示不做更改。

（2）八进制数：

```
chmod [选项] …八进制权限 文件…
```

其中，八进制权限是指要更改后的文件权限。

3）选项参数

chmod 主要选项参数如表 2.16 所示。

表 2.16　chmod 命令常见参数列表

选　　项	参 数 含 义
-c	若该文件权限确实已经更改，才显示其更改动作
-f	若该文件权限无法被更改，也不要显示错误信息
-v	显示权限变更的详细资料

华清远见

4）使用实例

chmod 涉及文件的访问权限，在此对相关的概念进行简单的回顾。

在 1.6.1 节中已经提到，文件的访问权限可表示成：**- rwx rwx rwx**。在此设有 3 种不同的访问权限：读（r）、写（w）和运行（x）；3 个不同的用户级别：文件拥有者（u）、所属的用户组（g）和系统中的其他用户（o）。在此，可增加一个用户级别 a（all）来表示所有的用户级别。

（1）对于第一种符号连接方式的 chmod 命令中，用加号“+”代表增加权限，用减号“−”代表删除权限，用等于号“=”代表设置权限。

例如，原来笔者系统中有文件 uClinux20031103.tgz，其权限如下：

```
[root@www linux]# ls -l
-rw-r--r--    1 root     root     79708616 Mar 24  2005 uClinux20031103.tgz
[root@www linux]# chmod a+rx,u+w uClinux20031103.tgz
[root@www linux]# ls -l
-rwxr-xr-x    1 root     root     79708616 Mar 24  2005 uClinux20031103.tgz
```

可见，在执行了 chmod 命令之后，文件拥有者除了拥有所有用户都有的可读和执行的权限外，还有可写的权限。

（2）对于第二种八进制数指定的方式，将文件权限字符代表的有效位设为“1”，即“rw-”、“rw-”和“r--”的八进制表示为“110”、“110”、“100”，把这个二进制串转换成对应的八进制数就是 6、6、4，也就是说该文件的权限为 664（3 位八进制数）。这样对于转化后八进制数、二进制及对应权限的关系如表 2.17 所示。

表 2.17　转化后八进制数、二进制及对应权限的关系

转换后八进制数	二　进　制	对 应 权 限	转换后八进制数	二　进　制	对 应 权 限
0	000	没有任何权限	4	100	只读
1	001	只能执行	5	101	只读和执行
2	010	只写	6	110	读和写
3	011	只写和执行	7	111	读、写和执行

同上例，原来笔者系统中有文件 genromfs-0.5.1.tar.gz，其权限如下：

```
[root@www linux]# ls -l
-rw-rw-r--    1 linux     linux        20543 Dec 29  2004 genromfs-0.5.1.tar.gz
[root@www linux]# chmod 765 genromfs-0.5.1.tar.gz
[root@www linux]# ls -l
-rwxrw-r-x    1 linux     linux        20543 Dec 29  2004 genromfs-0.5.1.tar.gz
```

可见，在执行了 chmod 765 命令之后，该文件的拥有者权限、文件组权限和其他用户权限都恰当地对应了。

5）使用说明

使用 chmod 必须具有 root 权限。

8．grep

华清远见 HQYJ.COM　华清远见教育集团官网：www.hqyj.com

1）作用

在指定文件中搜索特定的内容，并将含有这些内容的行标准输出。

2）格式

```
grep [选项] 格式 [文件及路径]
```

其中，格式是指要搜索的内容格式，若默认“文件及路径”则默认表示在当前目录下搜索。

3）常见参数

grep 主要选项参数如表 2.18 所示。

表 2.18　grep 命令常见参数列表

选　　项	参 数 含 义
-c	只输出匹配行的计数
-I	不区分大小写（只适用于单字符）
-h	查询多文件时不显示文件名
-l	查询多文件时只输出包含匹配字符的文件名
-n	显示匹配行及行号
-s	不显示不存在或无匹配文本的错误信息
-v	显示不包含匹配文本的所有行

4）使用实例

```
[root@www linux]# grep "hello" / -r
Binary file ./iscit2005/备份/iscit2004.sql matches
./ARM_TOOLS/uClinux-Samsung/linux-2.4.x/Documentation/s390/Debugging390.txt:hello world$2 = 0
…
```

该本例中，“hello”是要搜索的内容；“/ -r”是指定文件，表示搜索根目录下的所有文件。

5）使用说明

（1）在默认情况下，“grep”只搜索当前目录。如果此目录下有许多子目录，“grep”会以如下形式列出：“grep:sound:Is a directory”，这会使“grep”的输出难于阅读。但有两种解决的方法：

① 明确要求搜索子目录：grep –r（正如上例中所示）。

② 忽略子目录：grep -d skip。

（2）当预料到有许多输出时，可以通过管道将其转到“less”（分页器）上阅读，如 grep "h" ./ -r |less 分页阅读。

（3）grep 的特殊用法：

华清远见 HQYJ.COM　华清远见教育集团官网：www.hqyj.com

① grep pattern1|pattern2 files：显示匹配 pattern1 或 pattern2 的行。

② grep pattern1 files|grep pattern2：显示既匹配 pattern1 又匹配 pattern2 的行。

9. find

1）作用

在指定目录中搜索文件，它的使用权限是所有用户。

2）格式

```
find [路径][选项][描述]
```

其中，路径为文件搜索路径，系统开始沿着此目录树向下查找文件。它是一个路径列表，相互用空格分离。若默认路径，那么默认为当前目录；描述是匹配表达式，是 find 命令接受的表达式。

3）常见参数

（1）[选项]主要参数如表 2.19 所示。

表 2.19　find[选项]常见参数列表

选　　项	参 数 含 义
-depth	使用深度级别的查找过程方式，在某层指定目录中优先查找文件内容
-mount	不在其他文件系统（如 MSDOS、VFAT 等）的目录和文件中查找

（2）[描述]主要参数如表 2.20 所示。

表 2.20　find[描述]常见参数列表

选　　项	参 数 含 义
-name	支持通配符*和?
-user	用户名：搜索文件属主为用户名（ID 或名称）的文件
-print	输出搜索结果，并且打印

4）使用实例

```
[root@www linux]# find ./ -name qiong*.c
./qiong1.c
./iscit2005/qiong.c
```

在该实例中使用了“-name”选项支持通配符。

5）使用说明

（1）若使用目录路径为“/”，通常需要查找较多的时间，可以指定更为确切的路径以减少查找时间。

华清远见

（2）find 命令可以使用混合查找的方法。例如，要想在/etc 目录中查找大于 500 000 字节，并且在 24 小时内修改的某个文件，则可以使用-and（与）把两个查找参数链接起来组合成一个混合的查找方式，如“find /etc -size +500000c -and -mtime +1”。

10．locate

1）作用

用于查找文件。其方法是先建立一个包括系统内所有文件名称及路径的数据库，之后在寻找时就只需查询这个数据库，而不必实际深入档案系统之中了，因此其速度比 find 快很多。

2）格式

```
locate [选项]
```

3）常见参数

locate 主要选项参数如表 2.21 所示。

表 2.21　locate 命令常见参数列表

选　　项	参 数 含 义
-u	从根目录开始建立数据库
-U	指定的开始的位置建立数据库
-f	将特定的文件系统排除在数据库外，如 proc 文件系统中的文件
-r	使用正则运算式做寻找的条件
-o	指定数据库的名称

4）使用实例

```
[root@www linux]# locate issue -U ./
[root@www linux]# updatedb
[root@www linux]# locate -r issue*
./ARM_TOOLS/uClinux-Samsung/lib/libpam/doc/modules/pam_issue.sgml
./ARM_TOOLS/uClinux-Samsung/lib/libpam/modules/pam_issue
./ARM_TOOLS/uClinux-Samsung/lib/libpam/modules/pam_issue/Makefile
./ARM_TOOLS/uClinux-Samsung/lib/libpam/modules/pam_issue/pam_issue.c
…
```

实例中，首先在当前目录下建立了一个数据库，并且在更新了数据库之后进行正则匹配查找。通过运行可以发现 locate 的运行速度非常快。

5）使用说明

华清远见

华清远见 HQYJ.COM 华清远见教育集团官网：www.hqyj.com

locate 命令所查询的数据库是由 updatedb 程序来更新的，而 updatedb 是由 cron daemon 周期性建立的。但若找到的档案是最近才建立或刚更名的，可能会找不到，因为 updatedb 默认每天运行一次，用户可以通过修改 crontab（etc/crontab）来更新周期值。

11. ln

1）作用

为某一个文件在另外一个位置建立一个符号链接。当需要在不同的目录用到相同的文件时，Linux 允许用户不必在每一个需要的目录下都存放一个相同的文件，而只需将其他目录下的文件用 ln 命令链接即可，这样就不必重复占用磁盘空间。

2）格式

```
ln[选项] 目标 目录
```

3）常见参数

–s：建立符号链接（这也是通常情况下唯一使用的参数）。

4）使用实例

```
[root@www uclinux]# ln -s ../genromfs-0.5.1.tar.gz ./hello
[root@www uclinux]# ls -l
total 77948
lrwxrwxrwx   1 root    root    24 Jan 14 00:25 hello -> ../genromfs-0.5.1.tar.gz
```

该实例建立了当前目录的 hello 文件与上级目录之间的符号链接，可以看到，在 hello 的 ls –l 中的第一位为“l”，表示符号链接，同时还显示了链接的源文件。

5）使用说明

（1）ln 命令会保持每一处链接文件的同步性，也就是说，不论改动了哪一处，其他文件都会发生相同的变化。

（2）ln 的链接分为软链接和硬链接两种：

① **软链接**就是上面所说的 ln -s ** **，它只会在用户选定的位置上生成一个文件的镜像，而不会重复占用磁盘空间，平时使用较多的都是软链接。

② **硬链接**是不带参数的 ln ** **，它会在用户选定的位置上生成一个和源文件大小相同的文件。无论是软链接还是硬链接，文件都保持同步变化。

2.1.3 压缩打包相关命令

Linux 中打包压缩的相关命令如表 2.22 所示，本书以 gzip 和 tar 为例进行讲解。

表 2.22 Linux 常见打包压缩命令

命 令	命 令 含 义	格 式
bzip2	.bz2 文件的压缩（或解压）程序	bzip2[选项] 压缩（解压缩）的文件名
bunzip2	.bz2 文件的解压缩程序	bunzip2[选项] .bz2 压缩文件

华清远见

bzip2recover	用来修复损坏的.bz2 文件	bzip2recover .bz2 压缩文件
gzip	.gz 文件的压缩程序	gzip [选项] 压缩（解压缩）的文件名
gunzip	解压被 gzip 压缩过的文件	gunzip [选项] .gz 文件名
unzip	解压被 winzip 压缩过的.zip 文件	unzip [选项] .zip 压缩文件
compress	早期的压缩或解压程序（压缩后文件名为.Z）	compress [选项] 文件
tar	对文件目录进行打包或解包	tar [选项] [打包后文件名]文件目录列表

1．gzip

1）作用

对文件进行压缩和解压缩，而且 gzip 根据文件类型可自动识别压缩或解压。

2）格式

```
gzip [选项] 压缩（解压缩）的文件名
```

3）常见参数

gzip 主要选项参数如表 2.23 所示。

表 2.23　gzip 命令常见参数列表

选　项	参 数 含 义
-c	将输出信息写到标准输出上，并保留原有文件
-d	将压缩文件解压
-l	对每个压缩文件显示以下字段：压缩文件的大小、未压缩文件的大小、压缩比、未压缩文件的名称
-r	查找指定目录并压缩或解压缩其中的所有文件
-t	测试，检查压缩文件是否完整
-v	对每一个压缩和解压的文件，显示文件名和压缩比

4）使用实例

```
[root@www my]# gzip hello.c
[root@www my]# ls
hello.c.gz
[root@www my]# gzip -l hello.c
        compressed            uncompressed  ratio uncompressed_name
61                    39.3% hello.c
```

该实例将目录下的“hello.c”文件进行压缩，选项“-l”列出了压缩比。

5）使用说明

使用 gzip 压缩只能压缩单个文件，而不能压缩目录，其选项“-d”是将该目录下的所有文件逐个进行压缩，而不是压缩成一个文件。

2．tar

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

1）作用

对文件目录进行打包或解包。

在此需要对**打包**和**压缩**这两个概念进行区分。**打包**是指将一些文件或目录变成一个总的文件，而**压缩**则是将一个大的文件通过一些压缩算法变成一个小文件。为什么要区分这两个概念呢？这是由于在 Linux 中的很多压缩程序（如前面介绍的 gzip）只能针对一个文件进行压缩，这样当想要压缩较多的文件时，就要借助它的工具将这些文件先打成一个包，然后再用原来的压缩程序进行压缩。

2）格式

```
tar [选项] [打包后文件名]文件目录列表
```

tar 可自动根据文件名识别打包或解包动作，其中打包后文件名为用户自定义的打包后文件名称；文件目录列表可以是要进行打包备份的文件目录列表，也可以是进行解包的文件目录列表。

3）主要参数

tar 主要选项参数如表 2.24 所示。

表 2.24　tar 命令常见参数列表

选　项	参 数 含 义
-c	建立新的打包文件
-r	向打包文件末尾追加文件
-x	从打包文件中解出文件
-o	将文件解开到标准输出
-v	处理过程中输出相关信息
-f	对普通文件进行操作
-z	调用 gzip 来压缩打包文件，与-x 联用时调用 gzip 完成解压缩
-j	调用 bzip2 来压缩打包文件，与-x 联用时调用 bzip2 完成解压缩
-Z	调用 compress 来压缩打包文件，与-x 联用时调用 compress 完成解压缩

4）使用实例

```
[root@www home]# tar -cvf ycw.tar ./ycw
./ycw/
./ycw/.bash_logout
./ycw/.bash_profile
./ycw/.bashrc
./ycw/.bash_history
./ycw/my/
./ycw/my/1.c.gz
./ycw/my/my.c.gz
./ycw/my/hello.c.gz
./ycw/my/why.c.gz
[root@www home]# ls -l ycw.tar
```

华清远见

```
-rw-r--r--    1 root     root        10240 Jan 14 15:01 ycw.tar
```

该实例将./ycw目录下的文件加以打包，其中选项“-v”在屏幕上输出了打包的具体过程。

```
[root@www linux]# tar -zxvf linux-2.6.11.tar.gz
linux-2.6.11/
linux-2.6.11/drivers/
linux-2.6.11/drivers/video/
linux-2.6.11/drivers/video/aty/
…
```

该实例用选项“-z”调用gzip，并与-x联用时完成解压缩。

5）使用说明

tar命令除了用于常规的打包之外，使用更为频繁的是用选项“-z”或“-j”调用gzip或bzip2（Linux中另一种解压工具）完成对各种不同文件的解压。

表2.25对Linux中常见类型的文件解压命令做了总结。

表2.25　Linux常见类型的文件解压命令一览表

文件扩展名	解 压 命 令	示　例
.a	tar xv	tar xv hello.a
.z	Uncompress	uncompress hello.Z
.gz	Gunzip	gunzip hello.gz
.tar.Z	tar xvZf	tar xvZf hello.tar.Z
.tar.gz/.tgz	tar xvzf	tar xvzf hello.tar.gz
tar.bz2	tar jxvf	tar jxvf hello.tar.bz2
.rpm	安装：rpm –i	安装：rpm -i hello.rpm
	解压：rpm2cpio	解压：rpm2cpio hello.rpm
.deb（Debain中的文件格式）	安装：dpkg –i	安装：dpkg -i hello.deb
	解压：dpkg-deb --fsys-tarfile	解压：dpkg-deb --fsys-tarhello hello.deb
.zip	Unzip	unzip hello.zip

2.1.4　比较合并文件相关命令

1．diff

1）作用

比较两个不同的文件或不同目录下的两个同名文件，并生成补丁文件。

2）格式

```
diff[选项] 文件1 文件2
```


华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

diff 比较文件 1 和文件 2 的不同之处，并按照选项所指定的格式加以输出。diff 的格式分为命令格式和上下文格式，其中上下文格式又包括旧版上下文格式和新版上下文格式；命令格式分为标准命令格式、简单命令格式及混合命令格式，它们之间的区别会在“使用实例”中进行详细讲解。当选项默认时，diff 默认使用混合命令格式。

3）主要参数

diff 主要选项参数如表 2.26 所示。

表 2.26　diff 命令常见参数列表

选　　项	参 数 含 义
-r	对目录进行递归处理
-q	只报告文件是否有不同，不输出结果
-e，-ed	命令格式
-f	RCS（修订控制系统）命令简单格式
-c，--context	旧版上下文格式
-u，--unified	新版上下文格式
-Z	调用 compress 来压缩归档文件，与-x 联用时调用 compress 完成解压缩

4）使用实例

下面有两个文件 hello1.c 和 hello2.c：

```
//hello1.c
#include <stdio.h>
void main()
{
         printf("Hello!This is my home!\n");
}
//hello2.c
#include <stdio.h>
void main()
{
         printf("Hello!This is your home!\n");
}
```

以下实例主要讲解了各种不同格式的比较和补丁文件的创建方法。

（1）主要格式比较。首先使用**旧版上下文格式**进行比较。

```
[root@www ycw]# diff -c hello1.c hello2.c
*** hello1.c    Sat Jan 14 16:24:51 2006
--- hello2.c    Sat Jan 14 16:54:41 2006
***************
*** 1,5 ****
  #include <stdio.h>
```

华清远见

```
  void main()
  {
!       printf("Hello!This is my home!\n");
  }
--- 1,5 ----
  #include <stdio.h>
  void main()
  {
!       printf("Hello!This is your home!\n");
  }
```

可以看出，用旧版上下文格式进行输出时，在显示每个有差别行的同时还显示该行的上下 3 行，区别的地方用“！”加以标出。由于示例程序较短，上下 3 行已经包含了全部代码。

接着使用**新版上下文格式**进行比较。

```
[root@www ycw]# diff -u hello1.c hello2.c
--- hello1.c    Sat Jan 14 16:24:51 2006
+++ hello2.c    Sat Jan 14 16:54:41 2006
@@ -1,5 +1,5 @@
 #include <stdio.h>
 void main()
 {
-       printf("Hello!This is my home!\n");
+       printf("Hello!This is your home!\n");
 }
```

可以看出，在新版上下文格式输出时，仅把两个文件的不同之处分别列出，而相同之处没有重复列出，从而大大方便了用户的阅读。

接下来使用**命令格式**进行比较。

```
[root@www ycw]# diff -e hello1.c hello2.c
4c
        printf("Hello!This is your home!\n");
```

可以看出，命令格式输出时仅输出了不同的行，其中命令符“4c”中的数字表示行数，字母的含义为：**a——添加**，**b——删除**，**c——更改**。因此，“-e”选项的命令符表示：若要把 hello1.c 变为 hello2.c，只需把 hello1.c 的第 4 行改为显示出的“printf ("Hello!This is your home!\n")；”即可。

选项“-f”和选项“-e”显示的内容基本相同，就是数字和字母的顺序相交换了，从以下的输出结果可以看出。

```
[root@www ycw]# diff -f hello1.c hello2.c
c4
        printf("Hello!This is your home!\n");
```

在 diff 选项默认的情况下，输出结果如下：

```
[root@www ycw]# diff hello1.c hello2.c
4c4
<       printf("Hello!This is my home!\n");
```


华清远见 HQYJ.COM

华清远见教育集团官网：www.hqyj.com

```
---
>       printf("Hello!This is your home!\n");
```

可以看出，diff默认情况下的输出格式充分显示了如何将hello1.c转化为hello2.c的方法，即通过“4c4”实现。

（2）创建补丁文件（也就是差异文件）是diff的功能之一，不同的选项格式可以生成与之相对应的补丁文件如下所示。

```
[root@www ycw]# diff hello1.c hello2.c >hello.patch
[root@www ycw]# vi hello.patch
4c4
<       printf("Hello!This is my home!\n");
---
>       printf("Hello!This is your home!\n");
```

可以看出，使用默认选项创建补丁文件的内容和前面使用默认选项的输出内容是一样的。

2. patch

1）作用

该命令与diff配合使用，把生成的补丁文件应用到现有代码上。

2）格式

```
patch [选项] [待patch的文件[patch文件]]
```

常用的格式为：patch -pnum [patch 文件]，其中，-pnum是选项参数，在后面会详细介绍。

3）常见参数

patch主要选项参数如表2.27所示。

华清远见

表2.27 patch命令常见参数列表

选　项	参 数 含 义
-b	生成备份文件
-d	把dir设置为解释补丁文件名的当前目录
-e	把输入的补丁文件看做是ed脚本
-pnum	剥离文件名中的前NUM个目录成分
-t	在执行过程中不要求任何输入
-v	显示patch的版本号

下面对-punm选项进行说明。

首先查看以下示例（对分别位于xc.orig/config/cf/Makefile和xc.bsd/config/cf/Makefile的文件使用patch命令）。

```
diff -ruNa xc.orig/config/cf/Makefile xc.bsd/config/cf/Makefile
```

下面是 patch 文件的头标记：

```
--- xc.orig/config/cf/Imake.cf Fri Jul 30 12:45:47 1999
+++ xc.new/config/cf/Imake.cf Fri Jan 21 13:48:44 2000
```

这个 patch 如果直接应用，那么它会去找 xc.orig/config/cf 目录下的 Makefile 文件。假如用户源代码树的根目录是默认的 xc 而不是 xc.orig，则除了可以把 xc.orig 移到 xc 处之外，还有什么简单的方法应用此 patch 吗？NUM 就是为此而设的：patch 会把目标路径名剥去 NUM 个 "/"，也就是说，在此例中，-p1 的结果是 config/cf/Makefile，-p2 的结果是 cf/Makefile。因此，在此例中就可以用命令 cd xc;patch _p1 < /pathname/xxx.patch 完成操作。

4）使用实例

```
[root@www ycw]# diff hello1.c hello2.c >hello1.patch
[root@www ycw]# patch ./hello1.c < hello1.patch
patching file ./hello1.c
[root@www ycw] ]# vi hello1.c
#include <stdio.h>
void main()
{
        printf("Hello!This is your home!\n");
}
```

在该实例中，由于 patch 文件和源文件在同一目录下，因此直接给出了目标文件的目录。在应用了 patch 命令之后，hello1.c 的内容变为了 hello2.c 的内容。

5）使用说明

（1）如果 patch 失败，patch 命令会把成功的 patch 行补上其差异，同时（无条件）生成备份文件和一个.rej 文件。.rej 文件里是没有成功提交的 patch 行，需要手工打上补丁。这种情况在原码升级时有可能会发生。

（2）在多数情况下，patch 程序可以确定补丁文件的格式，当它不能识别时，可以使用 "-c"、"-e"、"-n" 或 "-u" 选项来指定输入的补丁文件的格式。由于只有 GNU patch 可以创建和读取新版上下文格式的 patch 文件，因此，除非能够确定补丁所面向的只是那些使用 GNU 工具的用户，否则，应该使用旧版上下文格式来生成补丁文件。

（3）为了使 patch 程序能够正常工作，需要上下文的行数至少是两行（即至少是有一处差别的文件）。

2.1.5　网络相关命令

Linux 下网络相关的常见命令如表 2.28 所示，本书仅以 ifconfig 和 ftp 为例进行说明。

表 2.28　Linux 下网络相关命令

选　　项	参 数 含 义	常见选项格式
netstat	显示网络连接、路由表和网络接口信息	netstat [-an]

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com



nslookup	查询一台机器的 IP 地址和其对应的域名	nslookup [IP 地址/域名]
finger	查询用户的信息	finger [选项] [使用者] [用户@主机]
ping	用于查看网络上的主机是否在工作	ping [选项] 主机名/IP 地址
ifconfig	查看和配置网络接口的参数	ifconfig [选项] [网络接口]
ftp	利用 ftp 协议上传和下载文件	在本节中会详细讲述
telnet	利用 Telnet 协议浏览信息	telent [选项] [IP 地址/域名]
ssh	利用 ssh 登录对方主机	ssh [选项] [IP 地址]

1. ifconfig

1）作用

用于查看和配置网络接口的地址和参数，包括 IP 地址、网络掩码、广播地址，它的使用权限是超级用户。

2）格式

ifconfig 有两种使用格式，分别用于查看和更改网络接口。

（1）ifconfig [选项] [网络接口]：用来查看当前系统的网络配置情况。

（2）ifconfig 网络接口 [选项] 地址：用来配置指定接口（如 eth0、eth1）的 IP 地址、网络掩码、广播地址等。

（3）常见参数

ifconfig 第二种格式的常见选项参数如表 2.29 所示。

表 2.29 ifconfig 命令选项常见参数列表

选　　项	参 数 含 义
-interface	指定的网络接口名，如 eth0 和 eth1
up	激活指定的网络接口卡
down	关闭指定的网络接口
broadcast address	设置接口的广播地址
poin to point	启用点对点方式
address	设置指定接口设备的 IP 地址
netmask address	设置接口的子网掩码

4）使用实例

首先，在本例中使用 ifconfig 的第一种格式来查看网口配置情况。

```
[root@linux workplace]# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:08:02:E0:C1:8A
          inet addr:59.64.205.70  Bcast:59.64.207.255  Mask:255.255.252.0
          inet6 addr: fe80::208:2ff:fee0:c18a/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:26931 errors:0 dropped:0 overruns:0 frame:0
          TX packets:3209 errors:0 dropped:0 overruns:0 carrier:0
```


华清远见 HQYJ.COM

华清远见教育集团官网：www.hqyj.com

```
              collisions:0 txqueuelen:1000
              RX bytes:6669382 (6.3 MiB)  TX bytes:321302 (313.7 KiB)
              Interrupt:11

lo            Link encap:Local Loopback
              inet addr:127.0.0.1  Mask:255.0.0.0
              inet6 addr: ::1/128 Scope:Host
              UP LOOPBACK RUNNING  MTU:16436  Metric:1
              RX packets:2537 errors:0 dropped:0 overruns:0 frame:0
              TX packets:2537 errors:0 dropped:0 overruns:0 carrier:0
              collisions:0 txqueuelen:0
              RX bytes:2093403 (1.9 MiB)  TX bytes:2093403 (1.9 MiB)
```

可以看出，使用 ifconfig 的显示结果中详细列出了所有活跃接口的 IP 地址、硬件地址、广播地址、子网掩码、回环地址等。

```
[root@linux workplace]# ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:08:02:E0:C1:8A
          inet addr:59.64.205.70  Bcast:59.64.207.255  Mask:255.255.252.0
          inet6 addr: fe80::208:2ff:fee0:c18a/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:27269 errors:0 dropped:0 overruns:0 frame:0
          TX packets:3212 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:6698832 (6.3 MiB)  TX bytes:322488 (314.9 KiB)
          Interrupt:11
```

在此例中，通过指定接口显示出对应接口的详细信息。另外，用户还可以通过指定参数“-a”来查看所有接口（包括非活跃接口）的信息。

接下来的示例指出了如何使用 ifconfig 的第二种格式来改变指定接口的网络参数配置。

```
[root@linux ~]# ifconfig eth0 down
[root@linux ~]# ifconfig
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:1931 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1931 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:2517080 (2.4 MiB)  TX bytes:2517080 (2.4 MiB)
```

在此例中，通过将指定接口的状态设置为 DOWN，暂停该接口的工作。

```
[root@linux workplace]# ifconfig eth0 210.25.132.142 netmask 255.255.255.0
[root@linux workplace]# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:08:02:E0:C1:8A
          inet addr:210.25.132.142  Bcast:210.25.132.255  Mask:255.255.255.0
          inet6 addr: fe80::208:2ff:fee0:c18a/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1722 errors:0 dropped:0 overruns:0 frame:0
          TX packets:5 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:147382 (143.9 KiB)  TX bytes:398 (398.0 b)
```


华清远见 HQYJ.COM 华清远见教育集团官网：www.hqyj.com




```
            Interrupt:11
…
```

从上例可以看出，ifconfig 改变了接口 eth0 的 IP 地址、子网掩码等，在之后的 ifconfig 查看中可以看出确实发生了变化。

5）使用说明

用 ifconfig 命令配置的网络设备参数不需重启就可生效，但在机器重新启动以后将会失效。

2. ftp

1）作用

该命令允许用户利用 FTP 协议上传和下载文件。

2）格式

```
ftp [选项] [主机名/IP]
```

ftp 相关命令包括使用命令和内部命令，其中使用命令的格式如上所列，主要用于登录到 ftp 服务器的过程中使用；内部命令是指成功登录后进行的一系列操作，下面会详细列出。若用户默认“主机名/IP”，则可在转入到 ftp 内部命令后继续选择登录。

3）常见参数

ftp 常见选项参数如表 2.30 所示。

表 2.30　ftp 选项常见参数列表

选　　项	参 数 含 义
-v	显示远程服务器的所有响应信息
-n	限制 ftp 的自动登录
-d	使用调试方式
-g	取消全局文件名

ftp 常见内部命令如表 2.31 所示。

表 2.31　ftp 常见内部命令

命　　令	命 令 含 义
account[password]	提供登录远程系统成功后访问系统资源所需的补充口令
Ascii	使用 ASCII 类型传输方式，为默认传输模式
bin/ type binary	使用二进制文件传输方式（嵌入式开发中的常见方式）
Bye	退出 ftp 会话过程
cd remote-dir	进入远程主机目录
Cdup	进入远程主机目录的父目录
chmod mode file-name	将远程主机文件 file-name 的存取方式设置为 mode

华清远见
HQYJ.COM
华清远见教育集团官网：www.hqyj.com

Close	中断与远程服务器的 ftp 会话（与 open 对应）
delete remote-file	删除远程主机文件
debug[debug-value]	设置调试方式，显示发送至远程主机的每条命令
dir/ls[remote-dir][local-file]	显示远程主机目录，并将结果存入本地文件 local-file
Disconnection	同 Close
get remote-file[local-file]	将远程主机的文件 remote-file 传至本地硬盘的 local-file

续表

命　令	命 令 含 义
lcd[dir]	将本地工作目录切换至 dir
mdelete[remote-file]	删除远程主机文件
mget remote-files	传输多个远程文件
mkdir dir-name	在远程主机中创建一个目录
mput local-file	将多个文件传输至远程主机
open host[port]	建立指定 ftp 服务器连接，可指定连接端口
Passive	进入被动传输方式（在这种模式下，数据连接是由客户程序发起的）
put local-file[remote-file]	将本地文件 local-file 传送至远程主机
reget remote-file[local-file]	类似于 get，但若 local-file 存在，则从上次传输中断处续传
size file-name	显示远程主机文件大小
System	显示远程主机的操作系统类型

4）使用实例

首先，在本例中使用 ftp 命令访问 ftp://study.byr.edu.cn 站点。

```
[root@linux ~]# ftp study.byr.edu.cn
Connected to study.byr.edu.cn.
220 Microsoft FTP Service
500 'AUTH GSSAPI': command not understood
500 'AUTH KERBEROS_V4': command not understood
KERBEROS_V4 rejected as an authentication type
Name (study.byr.edu.cn:root): anonymous
331 Anonymous access allowed, send identity (e-mail name) as password.
Password:
230 Anonymous user logged in.
Remote system type is Windows_NT.

ftp> dir
227 Entering Passive Mode (211,68,71,83,11,94).
125 Data connection already open; Transfer starting.
11-20-05  05:00PM       <DIR>          Audio
12-04-05  09:41PM       <DIR>          BUPT_NET_Material
01-07-06  01:38PM       <DIR>          Document
11-22-05  03:47PM       <DIR>          Incoming
```

华清远见

```
01-04-06  11:09AM       <DIR>          Material
226 Transfer complete.
```

以上使用ftp内部命令dir列出了在该目录下文件及目录的信息。

```
ftp> cd /Document/Wrox/Wrox.Beginning.SQL.Feb.2005.eBook-DDU
250 CWD command successful.
ftp> pwd
257 "/Document/Wrox/Wrox.Beginning.SQL.Feb.2005.eBook-DDU" is current directory.
```

以上实例通过cd命令进入相应的目录，可通过pwd命令进行验证。

```
ftp> lcd /root/workplace
Local directory now /root/workplace
ftp> get d-wbsq01.zip
local: d-wbsq01.zip remote: d-wbsq01.zip
200 PORT command successful.
150 Opening ASCII mode data connection for d-wbsq01.zip(1466768 bytes).
WARNING! 5350 bare linefeeds received in ASCII mode
File may not have transferred correctly.
226 Transfer complete.
1466768 bytes received in 1.7 seconds (8.6e+02 Kbytes/s)
```

接下来通过lcd命令首先改变用户的本地工作目录，也就是希望下载或上传的工作目录，接着通过get命令进行下载文件。由于ftp默认使用ASCII模式，因此，若希望改为其他模式如“bin”，直接输入bin即可，代码如下：

```
ftp> bin
200 Type set to I.
ftp> bye
221
```

最后使用bye命令退出ftp程序。

5）使用说明

（1）若需要匿名登录，则在“Name (**.**.**.**):”处输入anonymous，在“Password:”处输入自己的E-mail地址即可。

（2）若要传送二进制文件，务必要把模式改为bin。

2.2 Linux 系统服务



INIT进程的一个重要作用就是启动Linux系统服务（也就是运行在后台的守护进程）。Linux的系统服务包括两种：第一种是独立运行的系统服务，它们常驻内存中，自开机后一直启动着（如httpd），具有很快的响应速度；第二种是由xinet设定的服务。xinet能够同时监听多个指定的端口，在接受用户请求时，它能够根据用户请求的端口不同，启动不同的网络服务进程来处理这些用户请求。因此，可以把xinetd看做一个启动

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

服务的管理服务器，它决定把一个客户请求交给哪个程序处理，然后启动相应的守护进程。以下来分别介绍这两种系统服务。

2.2.1　独立运行的服务

独立运行的系统服务的启动脚本都放在/etc/init.d/目录中。如笔者系统中的系统服务的启动脚本如下：

```
[root@linux init.d]# ls /etc/rc.d/init.d
acpid dc_client iptables named pand rpcsvcgssd tux
anacron dc_server irda netdump pcmcia saslauthd vncserver
apmd diskdump irqbalance netfs portmap sendmail vsftpd
arptables_jf dovecot isdn netplugd psacct single watchquagga
atd dund killall network rawdevices smartd winbind
autofs firstboot kudzu NetworkManager readahead smb xfs
…
```

为了指定特定运行级别服务的开启或关闭，系统的各个不同运行级别都有不同的脚本文件，其目录为/etc/rcN.d，其中的 N 分别对应不用的运行级别。读者可以进入到各个不同的运行级别目录中查看相应服务的开启或关闭状态，如进入/rc3.d 目录中的文件如下：

```
[root@linux rc3.d]# ls /etc/rc3.d
K02NetworkManager K35winbind K89netplugd S10networ S28autofs S95anacron
K05saslauthd K36lisa K90bluetooth S12syslog S40smartd S95atd
K10dc_server K45named K94diskdump S13irqbalance S44acpid S97messagebus
K10psacct K50netdump K99microcode_ctl S13portmap S55cups S97rhnsd
…
```

可以看到，每个对应的服务都以“K”或“S”开头，其中，K 代表关闭（Kill），S 代表启动（Start），用户可以使用命令“+start|stop|status|restart”来对相应的服务进行操作。

在执行完相应的 rcN.d 目录下的脚本文件后，INIT 最后会执行 rc.local 来启动本地服务。因此，用户若想把某些非系统服务设置为自启动，可以编辑 rc.local 脚本文件，加上相应的执行语句即可。

另外，读者还可以使用命令“service+系统服务+操作”来方便地实现相应服务的操作，代码如下：

```
[root@linux xinetd.d]# service xinetd restart
停止 xinetd:                                        [  确定  ]
开启 xinetd:                                        [  确定  ]
```

2.2.2　xinetd 设定的服务

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

xinetd 管理系统中不经常使用的服务，这些服务程序只有在有请求时才由 xinetd 服务负责启动，一旦运行完毕服务自动结束。xinetd 的配置文件为/etc/xinetd.conf，它对 xinetd 的默认参数进行了配置：

```
#
# Simple configuration file for xinetd
#
# Some defaults, and include /etc/xinetd.d/
defaults
{
     instances = 60
     log_type = SYSLOG authpriv
     log_on_success = HOST PID
     log_on_failure = HOST
      cps = 25 30
}
includedir /etc/xinetd.d
```

从该配置文件的最后一行可以看出，xinetd 启动/etc/xinetd.d 为其配置文件目录。在对应的配置文件目录中可以看到每一个服务的基本配置，如 tftp 服务的配置脚本文件为：

```
service tftp
{
      socket_type = dgram//数据包格式
      protocol = udp//使用 UDP 传输
      wait = yes
      user = root
      server = /usr/sbin/in.tftpd
      server_args = -s /tftpboot
      disable= yes//不启动
      per_source  = 11
      cps = 100 2
      flags = IPv4
}
```

2.2.3 设定服务命令常用方法

设定系统服务除了在本节中提到的使用 service 之外，chkconfig 也是一个很好的工具，它能够为不同的系统级别设置不同的服务。

其常用格式如下：

（1）chkconfig --list（注意，如果没有 chkconfig 命令可以自己手动安装，在 list 前有两个小连线）：查看系统服务设定。

示例如下：

```
[root@linux xinetd.d]# chkconfig --list
sendmail          0:关闭  1:关闭  2:打开  3:打开  4:打开  5:打开  6:关闭
snmptrapd         0:关闭  1:关闭  2:关闭  3:关闭  4:关闭  5:关闭  6:关闭
gpm               0:关闭  1:关闭  2:打开  3:打开  4:打开  5:打开  6:关闭
syslog            0:关闭  1:关闭  2:打开  3:打开  4:打开  5:打开  6:关闭
…
```

（2）chkconfig--level N [服务名称] 指定状态：对指定级别指定系统服务。

```
[root@linux xinetd.d]# chkconfig --list|grep ntpd
ntpd                 0:关闭  1:关闭  2 关闭  3:关闭  4:关闭  5:关闭  6:关闭
[root@linux ~]# chkconfig --level 3 ntpd on
[root@linux ~]# chkconfig --list|grep ntpd
ntpd                 0:关闭  1:关闭  2:关闭  3:打开  4:关闭  5:关闭  6:关闭
```

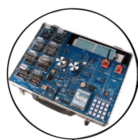
另外，在 2.1.1 节系统命令列表中指出的 setup 程序中也可以设定，而且是图形界面，操作较为方便，读者可以自行尝试。



2.3 本章习题

1．如何管理 Linux 系统用户？
2．如何列出系统中的隐藏文件？
3．如何复制整个目录？
4．怎样删除一个非空的目录？
5．如何创建一个链接？说明软链接和硬链接的区别。
6．在 Linux 系统中如何设置网络？

华清远见



华清远见
HQYJ.COM

第3章 嵌入式Linux编程环境

本章内容包括常用的Linux开发工具使用技巧和Linux编程技术，重点介绍常用的Linux编程工具和技巧。通过本章学习可以使读者快速掌握基本的Linux开发工具，为后续的嵌入式Linux开发打下基础。

华清远见

华清远见
HQYJ.COM
华清远见教育集团官网：www.hqyj.com

3.1 Linux 编辑器 vi 的使用



Linux 系统提供了一个完整的编辑器家族系列，如 ed、ex、vi 和 emacs 等。按功能可以分为两大类：行编辑器（ed、ex）和全屏幕编辑器（vi、emacs）。行编辑器每次只能对单行进行操作，使用起来很不方便，而全屏幕编辑器可以对整个屏幕进行编辑，用户编辑的文件直接显示在屏幕上，从而克服了行编辑的那种不直观的操作方式，便于用户学习和使用，具有强大的功能。

vi 是 Linux 系统的第一个全屏幕交互式编辑程序，它从诞生至今一直得到广大用户的青睐，历经数十年仍然是人们主要使用的文本编辑工具，足以见其生命力之强，而强大的生命力是其强大的功能带来的。由于大多数读者在此之前都已经用惯了 Windows 的 Word 等编辑器，因此，在刚刚接触时总会或多或少不适应，但只要习惯之后，就能感受到它的方便与快捷。

3.1.1 vi 的模式

vi 有 3 种模式，分别为命令行模式、插入模式及底行模式，下面具体介绍各模式的功能。

1．命令行模式

用户在用 vi 编辑文件时，最初进入的为一般模式。在该模式中可以通过上下移动光标进行“删除字符”或“整行删除”等操作，也可以进行“复制”、“粘贴”等操作，但无法编辑文字。

2．插入模式

只有在该模式下，用户才能进行文字编辑输入，用户可按 Esc 键回到命令行模式。

3．底行模式

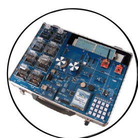
在该模式下，光标位于屏幕的底行。用户可以进行文件保存或退出操作，也可以设置编辑环境，如寻找字符串、列出行号等。

3.1.2 vi 的基本流程

（1）进入 vi，即在命令行下输入“vi hello（文件名）”，此时进入的是命令行模式，光标位于屏幕的上方，如图 3.1 所示。

（2）在命令行模式下输入“i”进入插入模式，如图 3.2 所示。可以看出，在屏幕底部显示有“插入”表示插入模式，在该模式下可以输入文字信息。

华清远见



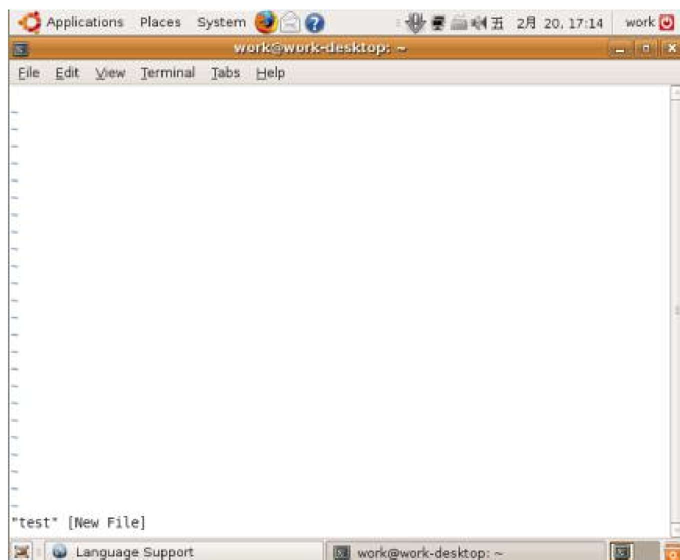


图 3.1　进入 vi 命令行模式

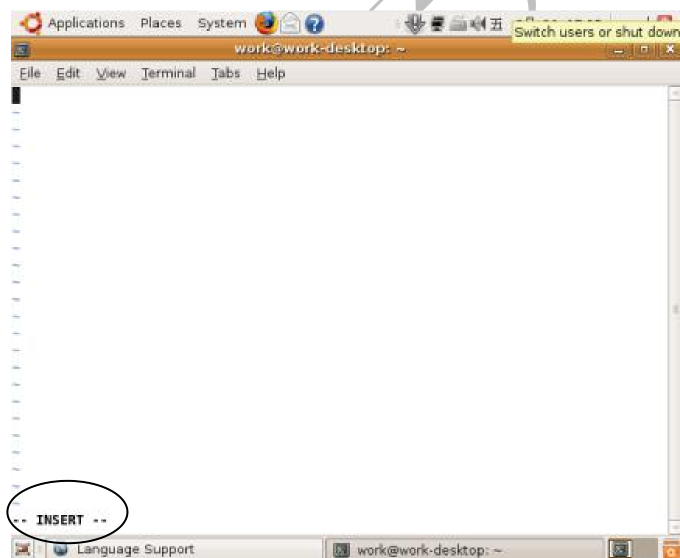


图 3.2　进入 vi 插入模式

（3）最后，在插入模式中按 Esc 键，则当前模式转入命令行模式，并在底行行中输入“:wq”（存盘退出）进入底行模式，如图 3.3 所示。

这样，就完成了一个简单的 vi 操作流程：命令行模式→插入模式→底行模式。由于 vi 在不同的模式下有不同的操作功能，因此，读者一定要时刻注意屏幕最下方的提示，分清所在的模式。

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

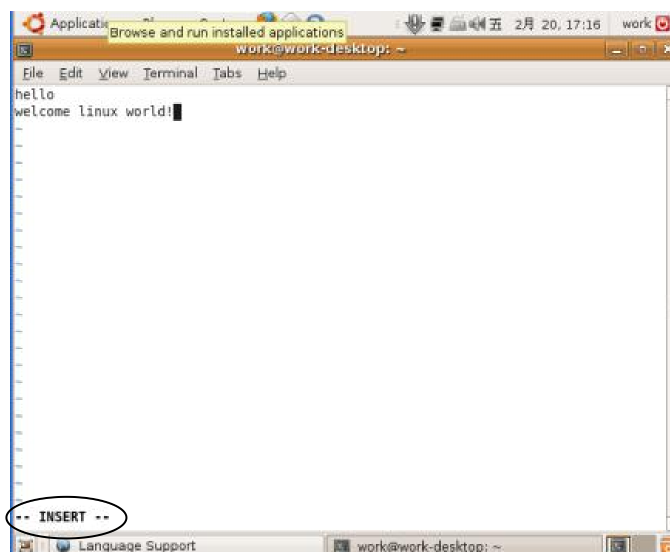


图 3.3 进入 vi 底行模式

3.1.3 vi 的各模式功能键

（1）命令行模式常见功能键如表 3.1 所示。

表 3.1 vi 命令行模式功能键

目 录	目 录 内 容
I	切换到插入模式，此时光标位于开始输入文件处
A	切换到插入模式，并从目前光标所在位置的下一个位置开始输入文字
O	切换到插入模式，且从行首开始插入新的一行
Ctrl+B	屏幕往“后”翻动一页
Ctrl+F	屏幕往“前”翻动一页
Ctrl+U	屏幕往“后”翻动半页
Ctrl+D	屏幕往“前”翻动半页
0（数字 0）	光标移动到本行的开头
G	光标移动到文章的最后
nG	光标移动到第 *n* 行
$	移动到光标所在行的“行尾”
n<Enter>	光标向下移动 *n* 行
/name	在光标之后查找一个名为 name 的字符串
?name	在光标之前查找一个名为 name 的字符串
X	删除光标所在位置的“后面”一个字符

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

续表

目　录	目 录 内 容
X	删除光标所在位置的“前面”一个字符
dd	删除光标所在行
ndd	从光标所在行开始向下删除 *n* 行
yy	复制光标所在行
nyy	复制光标所在行开始的向下 *n* 行
p	将缓冲区内的字符粘贴到光标所在位置（与 yy 搭配）
U	恢复前一个动作

（2）插入模式的功能键只有一个，也就是 Esc（退出到命令行模式）。

（3）底行模式常见功能键如表 3.2 所示。

表 3.2　vi 底行模式功能键

目　录	目 录 内 容
:w	将编辑的文件保存到磁盘中
:q	退出 vi（系统对做过修改的文件会给出提示）
:q!	强制退出 vi（对修改过的文件不做保存）
:wq	存盘后退出
:w [filename]	另存一个命为 filename 的文件
:set nu	显示行号，设定之后，会在每一行的前面显示对应行号
:set nonu	取消行号显示

华清远见

3.2 gcc 编译器



GNU CC（简称 gcc）是 GNU 项目中符合 ANSI C 标准的编译系统，能够编译用 C、C++和 Objective C 等语言编写的程序。gcc 不仅功能强大，而且可以编译如 C、C++、Objective C、Java、Fortran、Pascal、Modula-3 和 Ada 等多种语言；而且 gcc 又是一个交叉平台编译器，它能够在当前 CPU 平台上为多种不同体系结构的硬件平台开发软件，因此尤其适合在嵌入式领域的开发编译。本章中的示例除非特别注明，否则均采用 gcc 版本为 4.0.0。

如表 3.3 所示是 gcc 支持编译源文件的扩展名及其解释。

表 3.3　gcc 所支持扩展名解释

扩 展 名	所对应的语言	扩 展 名	所对应的语言
.c	C 原始程序	.s/.S	汇编语言原始程序
.C/.cc/.cxx	C++原始程序	.h	预处理文件（头文件）
.m	Objective- C 原始程序	.o	目标文件
.i	已经过预处理的 C 原始程序	.a/.so	编译后的库文件
.ii	已经过预处理的 C++原始程序		

3.2.1　gcc 编译流程解析

如本章开头提到的，gcc 的编译流程分为 4 个步骤，分别为：

（1）预处理（Pre-Processing）。

（2）编译（Compiling）。

（3）汇编（Assembling）。

（4）链接（Linking）。

下面就具体来查看一下 gcc 是如何完成这 4 个步骤的。

首先，有以下 hello.c 源代码：

```
#include<stdio.h>
int main()
{
     printf("Hello! This is our embedded world!\n");
     return 0;
}
```

1．预处理阶段

在该阶段，编译器将上述代码中的 stdio.h 编译进来，并且用户可以使用 gcc 的选项“-E”进行查看，该选项的作用是让 gcc 在预处理结束后停止编译过程。

```
[root@localhost Gcc]# gcc -E hello.c -o hello.i
```

在此处，选项“-o”是指目标文件；由表 3.3 可知，“.i”文件为已经过预处理的 C 原始程序。以下列出了 hello.i 文件的部分内容：

```
typedef int (*__gconv_trans_fct) (struct __gconv_step *,
        struct __gconv_step_data *, void *,
        __const unsigned char *,
        __const unsigned char **,
        __const unsigned char *, unsigned char **,
        size_t *);
…
# 2 "hello.c" 2
int main()
{
 printf("Hello! This is our embedded world!\n");
```


华清远见 HQYJ.COM　华清远见教育集团官网：www.hqyj.com

```
 return 0;
}
```

由此可见，gcc 确实进行了预处理，它把“stdio.h”的内容插入到 hello.i 文件中。

2．编译阶段

接下来进行的是编译阶段，在这个阶段中，gcc 首先要检查代码的规范性、是否有语法错误等，以确定代码实际要做的工作，在检查无误后，gcc 把代码翻译成汇编语言。用户可以使用“-S”选项来进行查看，该选项只进行编译而不进行汇编，生成汇编代码。

```
[root@localhost Gcc]# gcc –S hello.i –o hello.s
```

下面列出了 hello.s 的内容，可见 gcc 已经将其转化为汇编语言了，感兴趣的读者可以分析一下这一行简单的 C 语言程序是如何用汇编代码实现的。

```
    .file    "hello.c"
    .section .rodata
    .align 4
.LC0:
    .string       "Hello! This is our embedded world!"
    .text
.globl main
    .type main, @function
main:
    pushl %ebp
    movl %esp, %ebp
    subl $8, %esp
    andl $-16, %esp
    movl $0, %eax
    addl $15, %eax
    addl $15, %eax
    shrl $4, %eax
    sall $4, %eax
    subl %eax, %esp
    subl $12, %esp
    pushl $.LC0
    call puts
    addl $16, %esp
    movl $0, %eax
    leave
    ret
    .size    main, .-main
    .ident   "GCC: (GNU) 4.0.0 20050519 (Red Hat 4.0.0-8)"
    .section .note.GNU-stack,"",@progbits
```

3．汇编阶段

汇编阶段是把编译阶段生成的“.s”文件转成目标文件，读者在此使用选项“-c”就可看到汇编代码已转换为“.o”的二进制目标代码了，如下：

```
[root@localhost Gcc]# gcc –c hello.s –o hello.o
```

4．链接阶段

在成功编译之后，就进入了链接阶段。在这里涉及一个重要的概念——函数库。

读者可以重新查看这个小程序，在这个程序中并没有定义 printf 的函数实现，且在预编译中包含进的“stdio.h”中也只有该函数的声明，而没有定义函数的实现。那么，是在哪里实现“printf”函数的呢？最后的答案是：系统把这些函数实现都做到名为 libc.so.6 的库文件中去了，在没有特别指定时，gcc 会到系统默认的搜索路径/usr/lib 下进行查找，也就是链接到 libc.so.6 库函数中去，这样就能实现函数 printf 了，而这也就是链接的作用。

函数库一般分为静态库和动态库两种。静态库是指编译链接时，把库文件的代码全部加入到可执行文件中，因此生成的文件比较大，但在运行时也就不再需要库文件了，其后缀名一般为“.a”。动态库与之相反，在编译、链接时并没有把库文件的代码加入到可执行文件中，而是在程序执行时由运行时链接文件加载库，这样，可以节省系统的开销。动态库一般扩展名为“.so”，如前面所述的 libc.so.6 就是动态库。gcc 在编译时默认使用动态库。

完成了链接之后，gcc 就可以生成可执行文件，代码如下：

```
[root@localhost Gcc]# gcc hello.o -o hello
```

运行该可执行文件，出现正确的结果如下：

```
[root@localhost Gcc]# ./hello
Hello! This is our embedded world!
```

3.2.2 gcc 编译选项分析

gcc 有超过 100 个的可用选项，主要包括总体选项、告警和出错选项、优化选项和体系结构相关选项，以下对每一类中最常用的选项进行讲解。

1．总体选项

gcc 的总体选项如表 3.4 所示，很多在前面的示例中已经有所涉及。

表 3.4 gcc 总体选项列表

扩 展 名	所对应的语言
-c	只是编译不链接，生成目标文件“.o”
-S	只是编译不汇编，生成汇编代码
-E	只进行预编译，不做其他处理
-g	在可执行程序中包含标准调试信息
-o file	把输出文件输出到 file 中
-v	打印出编译器内部编译各过程的命令行信息和编译器的版本
-I dir	在头文件的搜索路径列表中添加 dir 目录

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

续表

扩 展 名	所对应的语言
-L dir	在库文件的搜索路径列表中添加 dir 目录
-static	链接静态库
-llibrary	链接名为 library 的库文件

对于“-c”、“-E”、“-o”、“-S”选项在 3.2.1 节中已经讲解了其使用方法，在此主要讲解另外两个非常常用的库依赖选项：“-I dir”和“-L dir”。

1）-I dir

正如表 3.4 中所述，“-I dir”选项可以在头文件的搜索路径列表中添加 dir 目录。由于 Linux 中头文件都默认放到了/usr/include/目录下，因此，当用户希望添加放置在其他位置的头文件时，就可以通过“-I dir”选项来指定，这样，gcc 就会到相应的位置查找对应的目录。

例如，在/root/workplace/Gcc 目录下有两个文件：

```
/*hello1.c*/
#include<my.h>
int main()
{
    printf("Hello!!\n");
    return 0;
}
/*my.h*/
#include<stdio.h>
```

这样，就可在 gcc 命令行中加入-I 选项：

```
[root@localhost Gcc] gcc hello1.c -I /root/workplace/Gcc/ -o hello1
```

这样，gcc 就能够执行出正确结果。

2）-L dir

选项“-L dir”的功能与“-I dir”类似，能够在库文件的搜索路径列表中添加 dir 目录。例如，有程序 hello_sq.c 需要用到目录/root/workplace/Gcc/lib 下的一个动态库 libsunq.so，则只需输入如下命令即可：

```
[root@localhost Gcc] gcc hello_sq.c -L /root/workplace/Gcc/lib -lsunq -o hello_sq
```

需要注意的是，“-I dir”和“-L dir”都只是指定了路径，而没有指定文件，因此不能在路径中包含文件名。

另外，值得详细解释一下的是“-l”选项，它指示 gcc 去链接库文件 libsunq.so。由于在 Linux 下的库文件命名时有一个规定：必须以 l、i、b 3 个字母开头，因此，在用“-l”选项指定链接的库文件名时可以省去 l、i、b 3 个字母。也就是说，gcc 在对“-lsunq”进行处理时，会自动去链接名为 libsunq.so 的文件。

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

2. 告警和出错选项

gcc 的告警和出错选项如表 3.5 所示。

表 3.5 gcc 告警和出错选项列表

选 项	含 义
-ansi	支持符合 ANSI 标准的 C 程序
-pedantic	允许发出 ANSI C 标准所列的全部警告信息
-pedantic-error	允许发出 ANSI C 标准所列的全部错误信息
-w	关闭所有告警
-Wall	允许发出 gcc 提供的所有有用的报警信息
-werror	把所有的告警信息转换为错误信息，并在告警发生时终止编译过程

下面结合实例对这几个告警和出错选项进行简单的讲解。

如有以下程序段：

```
#include<stdio.h>

void main()
{
    long long tmp = 1;
    printf("This is a bad code!\n");
    return 0;
}
```

这是一个很糟糕的程序，读者可以考虑一下有哪些问题。

1）-ansi

该选项强制 gcc 生成标准语法所要求的告警信息，尽管这还并不能保证所有没有警告的程序都是符合 ANSI C 标准的。运行结果如下：

```
[root@localhost Gcc]# gcc -ansi warning.c -o warning
warning.c: 在函数“main”中:
warning.c:7 警告：在无返回值的函数中，“return”带返回值
warning.c:4 警告：“main”的返回类型不是“int”
```

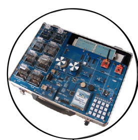
可以看出，该选项并没有发现“long long”这个无效数据类型的错误。

2）-pedantic

允许发出 ANSI C 标准所列的全部警告信息，同样也保证所有没有警告的程序都是符合 ANSI C 标准的。其运行结果如下：

```
[root@localhost Gcc]# gcc -pedantic warning.c -o warning
warning.c: 在函数“main”中:
warning.c:5 警告：ISO C90 不支持“long long”
warning.c:7 警告：在无返回值的函数中，“return”带返回值
warning.c:4 警告：“main”的返回类型不是“int”
```

可以看出，使用该选项查看出了“long long”这个无效数据类型的错误。



3）-Wall

允许发出 gcc 能够提供的所有有用的报警信息。该选项的运行结果如下：

```
[root@localhost Gcc]# gcc -Wall warning.c -o warning
warning.c:4 警告：“main”的返回类型不是“int”
warning.c: 在函数“main”中：
warning.c:7 警告：在无返回值的函数中，“return”带返回值
warning.c:5 警告：未使用的变量“tmp”
```

使用“-Wall”选项找出了未使用的变量 tmp，但它并没有找出无效数据类型的错误。另外，gcc 还可以利用选项对单独的常见错误分别指定警告。有关具体选项的含义，感兴趣的读者可以查看 gcc 手册进行学习。

3．优化选项

gcc 可以对代码进行优化，它通过编译选项“-On”来控制优化代码的生成，其中 n 是一个代表优化级别的整数。对于不同版本的 gcc 来讲，n 的取值范围及其对应的优化效果可能并不完全相同，比较典型的范围是从 0 变化到 2 或 3。

不同的优化级别对应不同的优化处理工作。例如，使用优化选项“-O”主要进行线程跳转（Thread Jump）和延迟退栈（Deferred Stack Pops）两种优化；使用优化选项“-O2”除了完成所有“-O1”级别的优化之外，同时还要进行一些额外的调整工作，如处理器指令调度等；选项“-O3”则还包括循环展开和其他一些与处理器特性相关的优化工作。

虽然优化选项可以加速代码的运行速度，但对于调试而言将是一个很大的挑战。因为代码在经过优化之后，原先在源程序中声明和使用的变量很可能不再使用，控制流也可能会突然跳转到意外的地方，循环语句也有可能因为循环展开而变得到处都有，所有这些对调试来讲都将是一场噩梦。所以，笔者建议在调试时最好不要使用任何优化选项，只有当程序在最终发行时才考虑对其进行优化。

华清远见

4．体系结构相关选项

gcc 的体系结构相关选项如表 3.6 所示。

表 3.6　gcc 体系结构相关选项列表

选　　项	含　　义
-mcpu=type	针对不同的 CPU 使用相应的 CPU 指令，可选择的 type 有 i386、i486、pentium 及 i686 等
-mieee-fp	使用 IEEE 标准进行浮点数的比较
-mno-ieee-fp	不使用 IEEE 标准进行浮点数的比较
-msoft-float	输出包含浮点库调用的目标代码
-mshort	把 int 类型作为 16 位处理，相当于 short int
-mrtd	强行将函数参数个数固定的函数用 ret NUM 返回，节省调用函数的一条指令

这些体系结构相关选项在嵌入式设计中会有较多的应用，读者需根据不同体系结构将对应的选项进行组合处理。在本书后面涉及具体实例会有针对性的讲解。

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com



3.3 gdb 调试器

调试是所有程序员都会面临的问题，如何提高程序员的调试效率，更好、更快地定位程序中的问题，从而加快程序开发的进度，是大家共同面对的。就如读者熟知的 Windows 下的一些调试工具，如 VC 自带的设置断点、单步跟踪等，都受到了广大用户的赞赏。那么，在 Linux 下有什么很好的调试工具呢？

本文所介绍的 gdb 调试器是一款 GNU 开发组织并发布的 UNIX/Linux 下的程序调试工具。虽然它没有图形化的友好界面，但是它强大的功能也足以与微软的 VC 工具等媲美。下面就请跟随笔者一步步学习 gdb 调试器。

3.3.1 gdb 使用流程

这里给出了一个短小的程序，由此带领读者熟悉一下 gdb 的使用流程，建议读者能够实际动手操作。

首先，打开 Linux 下的编辑器 vi，编辑如下代码（由于为了更好地熟悉 gdb 的操作，笔者在此使用 vi 编辑，希望读者能够参见 3.3 节中对 vi 的介绍，并熟练使用 vi）：

```
/*test.c*/
#include <stdio.h>
int sum(int m);
int main()
{
      int i,n=0;
      sum(50);
      for(i=1; i<=50; i++)
       {
         n += i;
       }
      printf("The sum of 1-50 is %d \n", n );

}
int sum(int m)
{
         int i,n=0;
         for(i=1; i<=m;i++)
            n += i;
         printf("The sum of 1-m is %d\n", n);
}
```

在保存退出后首先使用 gcc 对 test.c 进行编译。注意，一定要加上选项-g，这样编译出的可执行代码中才包含调试信息；否则，之后 gdb 无法载入该可执行文件。

```
[root@localhost Gdb]# gcc -g test.c -o test
```


华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

虽然这段程序没有错误，但调试完全正确的程序可以更加了解 gdb 的使用流程。接下来就启动 gdb 进行调试。注意，gdb 进行调试的是可执行文件，而不是如.c 的源代码，因此，需要先通过 gcc 编译生成可执行文件才能用 gdb 进行调试。

```
[root@localhost Gdb]# gdb test
GNU Gdb Red Hat Linux (6.3.0.0-1.21rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...Using host libthread_db
library "/lib/libthread_db.so.1".
(gdb)
```

可以看出，在 gdb 的启动画面中指出了 gdb 的版本号、使用的库文件等信息，接下来就进入了由“(gdb)”开头的命令行界面。

（1）查看文件。在 gdb 中输入“l”（list）就可以查看所载入的文件，代码如下：

```
(gdb) l
1       #include <stdio.h>
2       int sum(int m);
3       int main()
4       {
5             int i,n=0;
6             sum(50);
7             for(i=1; i<=50; i++)
8              {
9                n += i;
10             }
(gdb) l
11            printf("The sum of 1~50 is %d \n", n );
12
13      }
14      int sum(int m)
15      {
16                 int i,n=0;
17                 for(i=1; i<=m;i++)
18                      n += i;
19                 printf("The sum of 1~m is = %d\n", n);
20      }
```

可以看出，gdb 列出的源代码中明确地给出了对应的行号，这样就可以大大地方便代码的定位。

（2）设置断点。设置断点是调试程序中一个非常重要的手段，它可以使程序到一定位置暂停它的运行。因此，程序员在该位置处可以方便地查看变量的值、堆栈情况等，从而找出代码的症结所在。

在 gdb 中设置断点非常简单，只需在“b”后加入对应的行号即可（这是最常用的方式，另外还有其他方式设置断点），代码如下：

```
(gdb) b 6
Breakpoint 1 at 0x804846d: file test.c, line 6.
```

要注意的是，在 gdb 中利用行号设置断点是指代码运行到对应行之前将其停止，如上例中，代码运行到第 5 行之前暂停（并没有运行第 5 行）。

（3）查看断点情况。在设置完断点之后，用户可以输入“info b”来查看设置断点情况，在 gdb 中可以设置多个断点。

```
(gdb) info b
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x0804846d in main at test.c:6
```

（4）运行代码。接下来就可运行代码了，gdb 默认从首行开始运行代码，输入“r”（run）即可（若想从程序中指定行开始运行，可在 r 后面加上行号）。

```
(gdb) r
Starting program: /root/workplace/Gdb/test
Reading symbols from shared object read from target memory...done.
Loaded system supplied DSO at 0x5fb000

Breakpoint 1, main () at test.c:6
6                 sum(50);
```

可以看到，程序运行到断点处就停止了。

（5）查看变量值。在程序停止运行之后，程序员所要做的工作是查看断点处的相关变量值。在 gdb 中只需输入“p+变量值”即可，代码如下：

```
(gdb) p n
$1 = 0
(gdb) p i
$2 = 134518440
```

在此处，为什么变量“i”的值为如此奇怪的一个数字呢？原因就在于程序是在断点设置的对应行之前停止的，那么在此时并没有把“i”的数值赋为零，而只是一个随机的数字。但变量“n”是在第 4 行赋值的，故在此时已经为零。

（6）单步运行。单步运行可以使用命令“n”（next）或“s”（step），它们之间的区别在于：若有函数调用的时候，“s”会进入该函数而“n”不会进入该函数。因此，“s”就类似于 VC 等工具中的“step in”，“n”类似于 VC 等工具中的“step over”。它们的使用如下：

```
(gdb) n
The sum of 1-m is 1275
7           for(i=1; i<=50; i++)
(gdb) s
sum (m=50) at test.c:16
16              int i,n=0;
```

可见，使用“n”后，程序显示函数 sum 的运行结果并向下执行，而使用“s”后则进入到 sum 函数之中单步运行。

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

（7）恢复程序运行。在查看完所需变量及堆栈情况后，就可以使用命令“c”（continue）恢复程序的正常运行。这时，它会把剩余还未执行的程序执行完，并显示剩余程序中的执行结果。以下是之前使用“n”命令恢复后的执行结果：

```
(gdb) c
Continuing.
The sum of 1-50 is :1275

Program exited with code 031.
```

可以看出，程序在运行完后退出，之后程序处于“停止状态”。

3.3.2 gdb 基本命令

gdb 的命令可以通过查看帮助进行查找。由于 gdb 的命令很多，因此 gdb 的帮助将其分成了很多种类（class），用户可以通过进一步查看相关 class 找到相应命令，如下：

```
(gdb) help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
…
Type "help" followed by a class name for a list of commands in that class.
Type "help" followed by command name for full documentation.
Command name abbreViations are allowed if unambiguous.
```

上述列出了 gdb 各个分类的命令，注意底部的加粗部分说明其为分类命令。接下来可以具体查找各分类种的命令，如下：

```
(gdb) help data
Examining data.

List of commands:

call -- Call a function in the program
delete display -- Cancel some expressions to be displayed when program stops
delete mem -- Delete memory region
disable display -- Disable some expressions to be displayed when program stops
…
Type "help" followed by command name for full documentation.
Command name abbreViations are allowed if unambiguous.
```

至此，若用户想要查找 call 命令，就可输入“help call”。

```
(gdb) help call
Call a function in the program.
The argument is the function name and arguments, in the notation of the
current working language.  The result is printed and saved in the value
```

```
history, if it is not void.
```

当然，若用户已知命令名，直接输入“help [command]”也是可以的。

gdb 中的命令主要分为以下几类：工作环境相关命令、设置断点与恢复命令、源代码查看命令、查看运行数据相关命令及修改运行参数命令，以下就分别对这几类命令进行讲解。

1．工作环境相关命令

gdb 中不仅可以调试所运行的程序，而且还可以对程序相关的工作环境进行相应的设定，甚至还可以使用 Shell 中的命令进行相关的操作，其功能极其强大。如表 3.7 所示为 gdb 常见工作环境相关命令。

表 3.7　gdb 工作环境相关命令

命 令 格 式	含　义
set args	指定运行时的参数，如 set args 2
show args	查看设置好的运行参数
path dir	设定程序的运行路径
show paths	查看程序的运行路径
set enVironment var [=value]	设置环境变量
show enVironment [var]	查看环境变量
cd dir	进入到 dir 目录，相当于 Shell 中的 cd 命令
pwd	显示当前工作目录
shell command	运行 Shell 的 command 命令

2．设置断点与恢复命令

gdb 中设置断点与恢复的常见命令如表 3.8 所示。

表 3.8　gdb 设置断点与恢复相关命令

命 令 格 式	含　义
bnfo b	查看所设断点
break 行号或函数名 <条件表达式>	设置断点
tbreak 行号或函数名 <条件表达式>	设置临时断点，到达后被自动删除
delete [断点号]	删除指定断点，其断点号为“info b”中的第一栏。若默认断点号则删除所有断点
disable [断点号]	停止指定断点，使用“info b”仍能查看此断点。同 delete 一样，默认断点号则停止所有断点
enable [断点号]	激活指定断点，即激活被 disable 停止的断点
condition [断点号] <条件表达式>	修改对应断点的条件
ignore [断点号]<num>	在程序执行中，忽略对应断点 num 次

华清远见 HQYJ.COM

华清远见教育集团官网：www.hqyj.com

续表

命令格式	含　义
step	单步恢复程序运行，且进入函数调用
next	单步恢复程序运行，但不进入函数调用
finish	运行程序，直到当前函数完成返回
c	继续执行函数，直到函数结束或遇到新的断点

由于设置断点在 gdb 的调试中非常重要，所以在此着重讲解一下 gdb 中设置断点的方法。

gdb 中设置断点有多种方式：其一是按行设置断点，设置方法在 3.3.1 节已经指出，在此就不重复了；另外，还可以设置函数断点和条件断点，在此结合 3.3.1 节中的代码，具体介绍后两种设置断点的方法。

1）函数断点

gdb 中按函数设置断点只需把函数名列在命令“b”之后，代码如下：

```
(gdb) b sum
Breakpoint 1 at 0x80484ba: file test.c, line 16.
(gdb) info b
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x080484ba in sum at test.c:16
```

要注意的是，此时的断点实际是在函数的定义处，也就是在 16 行处（注意，第 16 行还未执行）。

2）条件断点

gdb 中设置条件断点的格式为：

```
b 行数
```

或

```
函数名 if 表达式
```

具体实例如下：

```
(gdb) b 8 if i==10
Breakpoint 1 at 0x804848c: file test.c, line 8.
(gdb) info b
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x0804848c in main at test.c:8
        stop only if i == 10
(gdb) r
Starting program: /home/yul/test
The sum of 1-m is 1275

Breakpoint 1, main () at test.c:9
9              n += i;
(gdb) p i
$1 = 10
```


华清远见 HQYJ.COM　华清远见教育集团官网：www.hqyj.com

可以看到，该例中在第 8 行（也就是运行完第 7 行的 for 循环）设置了一个“i==10”的条件断点，在程序运行之后可以看出，程序确实在 i 为 10 时暂停运行。

3．gdb 中源代码查看相关命令

在 gdb 中可以查看源代码以方便其他操作，其常见的相关命令如表 3.9 所示。

表 3.9　gdb 源代码查看相关命令

命 令 格 式	含　　义
list <行号>|<函数名>	查看指定位置代码
file [文件名]	加载指定文件
forward-search 正则表达式	源代码前向搜索
reverse-search 正则表达式	源代码后向搜索
dir dir	停止路径名
show directories	显示定义了的源文件搜索路径
info line	显示加载到 gdb 内存中的代码

4．gdb 中查看运行数据相关命令

gdb 中查看运行数据是指当程序处于“运行”或“暂停”状态时，可以查看的变量及表达式的信息，其常见命令如表 3.10 所示。

表 3.10　gdb 查看运行数据相关命令

命 令 格 式	含　　义
print 表达式|变量	查看程序运行时对应表达式和变量的值
x <n/f/u>	查看内存变量内容，其中 n 为整数表示显示内存的长度，f 表示显示的格式，u 表示从当前地址往后请求显示的字节数
display 表达式	设定在单步运行或其他情况中，自动显示的对应表达式的内容

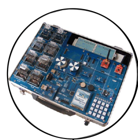
5．gdb 中修改运行参数相关命令

gdb 还可以修改运行时的参数，并使该变量按照用户当前输入的值继续运行。它的设置方法为：在单步执行的过程中输入命令“set 变量＝设定值”。这样，在此之后，程序就会按照该设定的值运行了。下面，笔者结合 3.3.1 节的代码将 n 的初始值设为 4，其代码如下：

```
(gdb) b 7
Breakpoint 5 at 0x804847a: file test.c, line 7.
(gdb) r
Starting program: /home/yul/test
The sum of 1-m is 1275

Breakpoint 5, main () at test.c:7
7               for(i=1; i<=50; i++)
(gdb) set n=4
(gdb) c
```


华清远见 HQYJ.COM　华清远见教育集团官网：www.hqyj.com
华清远见




```
Continuing.
The sum of 1-50 is 1279

Program exited with code 031.
```

可以看到，最后的运行结果确实比之前的值大了 4。

3.4 make 工程管理器



到目前为止，读者已经了解了如何在 Linux 下使用编辑器编写代码，如何使用 gcc 把代码编译成可执行文件，还学习了如何使用 gdb 来调试程序。那么，所有的工作看似已经完成了，为什么还需要 make 这个工程管理器呢？

所谓工程管理器，顾名思义，是用来管理较多的文件的工具。读者可以试想一下，有一个上百个文件的代码构成的项目，如果其中只有一个或少数几个文件进行了修改，按照之前所学的 gcc 编译工具，就不得不把这所有的文件重新编译一遍，因为编译器并不知道哪些文件是最近更新的，而只知道需要包含这些文件才能把源代码编译成可执行文件，于是，程序员就不得不再重新输入数目如此庞大的文件名以完成最后的编译工作。

人们就希望有一个工程管理器能够自动识别更新了的文件代码，同时又不需要重复输入冗长的命令行，这样，make 工程管理器也就应运而生了。

实际上，make 工程管理器也就是一个“自动编译管理器”，这里的“自动”是指它能够根据文件时间戳自动发现更新过的文件而减少编译的工作量，同时，它通过读入 Makefile 文件的内容来执行大量的编译工作，用户只需编写一次简单的编译语句就可以了。它大大提高了实际项目的工作效率，而且几乎所有 Linux 下的项目编程均会涉及它，希望读者能够认真学习本节内容。

华清远见

3.4.1 Makefile 基本结构

Makefile 是 make 读入的唯一配置文件，因此本节的内容实际就是讲述 Makefile 的编写规则。在一个 Makefile 中通常包含如下内容：

（1）需要由 make 工具创建的目标体（target），通常是目标文件或可执行文件。

（2）要创建的目标体所依赖的文件（dependency_file）。

（3）创建每个目标体时需要运行的命令（command）。

它的格式为：

```
target: dependency_files
        command
```

例如，有两个文件分别为 hello.c 和 hello.h，创建的目标体为 hello.o，执行的命令为 gcc 编译指令：gcc –c hello.c，那么，对应的 Makefile 就可以写为：

华清远见 HQYJ.COM

华清远见教育集团官网：www.hqyj.com

```
#The simplest example
hello.o: hello.c hello.h
        gcc -c hello.c -o hello.o
```

接着就可以使用 make 了。使用 make 的格式为：make target，这样 make 就会自动读入 Makefile（也可以是首字母小写 makefile）并执行对应 target 的 command 语句，并会找到相应的依赖文件，如下：

```
[root@localhost makefile]# make hello.o
gcc -c hello.c -o hello.o
[root@localhost makefile]# ls
hello.c  hello.h  hello.o  Makefile
```

可以看到，Makefile 执行了“hello.o”对应的命令语句，并生成了“hello.o”目标体。

3.4.2 Makefile 变量

上面示例的 Makefile 在实际中是几乎不存在的，因为它过于简单，仅包含两个文件和一个命令，在这种情况下完全没有必要编写 Makefile 而只需在 Shell 中直接输入即可。在实际中使用的 Makefile 往往是包含很多的文件和命令的，这也是 Makefile 产生的原因。下面就给出稍微复杂一些的 Makefile 进行讲解。

```
sunq:kang.o yul.o
Gcc kang.o bar.o -o myprog
kang.o : kang.c kang.h head.h
Gcc -Wall -O -g -c kang.c -o kang.o
yul.o : bar.c head.h
Gcc - Wall -O -g -c yul.c -o yul.o
```

在这个 Makefile 中有 3 个目标体（target），分别为 sunq、kang.o 和 yul.o，其中，第一个目标体的依赖文件就是后两个目标体。如果用户使用命令“make sunq”，则 make 管理器就是找到 sunq 目标体开始执行。

这时，make 会自动检查相关文件的时间戳。首先，在检查“kang.o”、“yul.o”和“sunq”3 个文件的时间戳之前，它会向下查找那些把“kang.o”或“yul.o”作为目标文件的时间戳。例如，“kang.o”的依赖文件为“kang.c”、“kang.h”、“head.h”，如果这些文件中任何一个的时间戳比“kang.o”新，则命令“gcc –Wall –O -g –c kang.c -o kang.o”将会执行，从而更新文件“kang.o”。在更新完“kang.o”或“yul.o”之后，make 会检查最初的“kang.o”、“yul.o”和“sunq”3 个文件，只要文件“kang.o”或“yul.o”中的任何文件时间戳比 sunq 新，则第二行命令就会被执行。这样，make 就完成了自动检查时间戳的工作，开始执行编译工作。这也就是 make 工作的基本流程。

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

接下来，为了进一步简化编辑和维护 Makefile，make 允许在 Makefile 中创建和使用变量。变量是在 Makefile 中定义的名字，用来代替一个文本字符串，该文本字符串称为该变量的值。在具体要求下，这些值可以代替目标体、依赖文件、命令及 Makefile 文件中的其他部分。在 Makefile 中的变量定义有两种方式：一种是递归展开方式，另一种是简单方式。

递归展开方式定义的变量是在引用该变量时进行替换的，即如果该变量包含了对其他变量的应用，则在引用该变量时一次性将内嵌的变量全部展开。虽然这种类型的变量能够很好地完成用户的指令，但是它也有严重的缺点，如不能在变量后追加内容（因为语句：CFLAGS = $(CFLAGS) -O 在变量扩展过程中可能导致无穷循环）等。

为了避免上述问题，简单扩展型变量的值在定义处展开，并且只展开一次，因此它不包含任何对其他变量的引用，从而消除变量的嵌套引用。

递归展开方式的定义格式为：

```
VAR=var
```

简单扩展方式的定义格式为：

```
VAR: =var
```

make 中的变量均使用格式为：

```
$(VAR)
```

下面给出了上例中用变量替换修改后的 Makefile，这里用 OBJS 代替 kang.o 和 yul.o，用 CC 代替 gcc，用 CFLAGS 代替“-Wall -O –g”。这样，在以后修改时，就可以只修改变量定义，而不需要修改下面的定义实体，从而大大简化了 Makefile 维护的工作量。

经变量替换后的 Makefile 如下：

```
OBJS = kang.o yul.o
CC = gcc
CFLAGS = -Wall -O -g
sunq : $(OBJS)
      $(CC) $(OBJS) -o sunq
kang.o : kang.c kang.h
      $(CC) $(CFLAGS) -c kang.c -o kang.o
yul.o : yul.c yul.h
      $(CC) $(CFLAGS) -c yul.c -o yul.o
```

可以看到，此处变量是以递归展开方式定义的。

Makefile 中的变量分为用户自定义变量、预定义变量、自动变量及环境变量。如上例中的 OBJS 就是用户自定义变量。自定义变量的值由用户自行设定，而预定义变量和自动变量为通常在 Makefile 都会出现的变量，其中部分有默认值，也就是常见的设定值，当然用户也可以对其进行修改。

预定义变量包含了常见编译器、汇编器的名称及其编译选项。表 3.11 列出了 Makefile 中常见预定义变量及其部分默认值。

表 3.11　Makefile 中常见预定义变量

变　　量	含　　义

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

续表

变　　量	含　　义
AR	库文件维护程序的名称，默认值为 ar
AS	汇编程序的名称，默认值为 as
CC	C 编译器的名称，默认值为 cc
CPP	C 预编译器的名称，默认值为$(CC) –E
CXX	C++编译器的名称，默认值为 g++
FC	FORTRAN 编译器的名称，默认值为 f77
RM	文件删除程序的名称，默认值为 rm –f
ARFLAGS	库文件维护程序的选项，无默认值
ASFLAGS	汇编程序的选项，无默认值
CFLAGS	C 编译器的选项，无默认值
CPPFLAGS	C 预编译的选项，无默认值
CXXFLAGS	C++编译器的选项，无默认值
FFLAGS	FORTRAN 编译器的选项，无默认值

可以看出，上例中的 CC 和 CFLAGS 是预定义变量，其中由于 CC 没有采用默认值，因此，需要把“CC=gcc”明确列出来。

由于常见的 gcc 编译语句中通常包含目标文件和依赖文件，而这些文件在 Makefile 文件中目标体的一行已经有所体现，因此，为了进一步简化 Makefile 的编写，引入了自动变量。自动变量通常可以代表编译语句中出现的目标文件和依赖文件等，并且具有本地含义（即下一语句中出现的相同变量代表的是下一语句的目标文件和依赖文件）。表 3.12 列出了 Makefile 中常见的自动变量。

表 3.12　Makefile 中常见自动变量

变　　量	含　　义
$*	不包含扩展名的目标文件名称
$+	所有的依赖文件，以空格分开，并以出现的先后为序，可能包含重复的依赖文件
$<	第一个依赖文件的名称
$?	所有时间戳比目标文件晚的依赖文件，并以空格分开
$@	目标文件的完整名称
$^	所有不重复的依赖文件，以空格分开
$%	如果目标是归档成员，则该变量表示目标的归档成员名称

自动变量的书写比较难记，但是在熟练了之后会非常方便，请读者结合下例中的自动变量改写的 Makefile 进行记忆。

```
OBJS = kang.o yul.o
CC = Gcc
```

华清远见

```
CFLAGS = -Wall -O -g
sunq : $(OBJS)
     $(CC) $^ -o $@
kang.o : kang.c kang.h
     $(CC) $(CFLAGS) -c $< -o $@
yul.o : yul.c yul.h
     $(CC) $(CFLAGS) -c $< -o $@
```

另外，在 Makefile 中还可以使用环境变量。使用环境变量的方法相对比较简单，make 在启动时会自动读取系统当前已经定义了的环境变量，并且会创建与之具有相同名称和数值的变量。但是，如果用户在 Makefile 中定义了相同名称的变量，那么用户自定义变量将会覆盖同名的环境变量。

3.4.3　Makefile 规则

Makefile 的规则是 make 进行处理的依据，它包括目标体、依赖文件及其之间的命令语句。一般，Makefile 中的一条语句就是一个规则。在上面的例子中，都显式地指出了 Makefile 中的规则关系，如“$(CC) $(CFLAGS) -c $< -o $@”，但为了简化 Makefile 的编写，make 还定义了隐式规则和模式规则，下面分别对其进行讲解。

1．隐式规则

隐式规则能够告诉 make 怎样使用传统的技术完成任务，这样，当用户使用它们时就不必详细指定编译的具体细节，而只需把目标文件列出即可。make 会自动搜索隐式规则目录来确定如何生成目标文件。如上例就可以写成：

```
OBJS = kang.o yul.o
CC = Gcc
CFLAGS = -Wall -O -g
sunq : $(OBJS)
     $(CC) $^ -o $@
```

为什么可以省略后两句呢？因为 make 的隐式规则指出：所有“.o”文件都可自动由“.c”文件使用命令“$(CC) $(CPPFLAGS) $(CFLAGS) -c file.c –o file.o”生成。这样，“kang.o”和“yul.o”就会分别调用“$(CC) $(CFLAGS) -c kang.c -o kang.o”和“$(CC) $(CFLAGS) -c yul.c -o yul.o”生成。

表 3.13 给出了常见的隐式规则目录。

表 3.13　Makefile 中常见隐式规则目录

对应语言扩展名	规　　则
C 编译：**.c** 变为**.o**	$(CC) –c $(CPPFLAGS) $(CFLAGS)
C++编译：**.cc** 或**.C** 变为**.o**	$(CXX) -c $(CPPFLAGS) $(CXXFLAGS)
Pascal 编译：**.p** 变为**.o**	$(PC) -c $(PFLAGS)
Fortran 编译：**.r** 变为**-o**	$(FC) -c $(FFLAGS)

华清远见 HQYJ.COM

华清远见教育集团官网：www.hqyj.com

2．模式规则

模式规则是用来定义相同处理规则的多个文件的。它不同于隐式规则，隐式规则仅仅能够用 make 默认的变量来进行操作，而模式规则还能引入用户自定义变量，为多个文件建立相同的规则，从而简化 Makefile 的编写。

模式规则的格式类似于普通规则，这个规则中的相关文件前必须用“%”标明。使用模式规则修改后的 Makefile 的编写如下：

```
OBJS = kang.o yul.o
CC = Gcc
CFLAGS = -Wall -O -g
sunq : $(OBJS)
     $(CC) $^ -o $@
%.o : %.c
     $(CC) $(CFLAGS) -c $< -o $@
```

3.4.4 make 管理器的使用

使用 make 管理器非常简单，只需在 make 命令的后面输入目标名即可建立指定的目标。如果直接运行 make，则建立 Makefile 中的第一个目标。

此外，make 还有丰富的命令行选项，可以完成各种不同的功能。表 3.14 列出了常用的 make 命令行选项。

表 3.14 make 的命令行选项

命 令 格 式	含 义
-C dir	读入指定目录下的 Makefile
-f file	读入当前目录下的 file 文件作为 Makefile
-i	忽略所有的命令执行错误
-I dir	指定被包含的 Makefile 所在目录
-n	只打印要执行的命令，但不执行这些命令
-p	显示 make 变量数据库和隐式规则
-s	在执行命令时不显示命令
-w	如果 make 在执行过程中改变目录，则打印当前目录名

华清远见

3.5 使用 autotools



在 3.4 节，读者已经了解到了 make 项目管理器的强大功能。的确，Makefile 可以帮助 make 完成它的使命，但要承认的是，编写 Makefile 确实不是一件轻松的事，尤其对于一个较大的项目而言更是如此。那么，有没有一种轻松的手段生成 Makefile，同时又能让用户享受 make 的优越性呢？本节要讲的 autotools 系列工具正是为此而设的，它只需用户输入简单的目标文件、依赖文件、文件目录等就可以轻松地生成 Makefile，这无疑是广大用户的所希望的。另外，这些工具还可以完成系统配置信息的收集，从而方便地处理各种移植性的问题。也正是基于此，现在 Linux 上的软件开发一般都用 autotools 来制作 Makefile。

3.5.1 autotools 使用流程

autotools 是系列工具，读者首先要确认系统是否装了以下工具（可以使用 which 命令进行查看）：

- aclocal。
- autoscan。
- autoconf。
- autoheader。
- automake。

使用 autotools 主要就是利用各个工具的脚本文件以生成最后的 Makefile。其总体流程如下：

（1）使用 aclocal 生成一个“aclocal.m4”文件，该文件主要处理本地的宏定义。

（2）改写“configure.scan”文件，将其重命名为“configure.in”，并使用 autoconf 文件生成 configure 文件。

接下来，将通过一个简单的 hello.c 例子熟悉 autotools 生成 Makefile 的过程。由于在这一过程中会涉及较多的脚本文件，为了更清楚地了解相互之间的关系，强烈建议读者实际动手操作以体会其整个过程。

1．autoscan

它会在给定目录及其子目录树中检查源文件，若没有给出目录，就在当前目录及其子目录树中进行检查。它会搜索源文件以寻找一般的移植性问题并创建一个文件“configure.scan”，该文件就是接下来 autoconf 要用到的“configure.in”原型，代码如下：

```
[root@localhost automake]# autoscan
autom4te: configure.ac: no such file or directory
autoscan: /usr/bin/autom4te failed with exit status: 1
[root@localhost automake]# ls
autoscan.log  configure.scan  hello.c
```

华清远见

由上述代码可知，autoscan 首先会尝试读入“configure.ac”（同 configure.in 的配置文件）文件，此时还没有创建该配置文件，于是它会自动生成一个“configure.in”的原型文件“configure.scan”。

2. autoconf

configure.in 是 autoconf 的脚本配置文件，它的原型文件“configure.scan”如下：

```
#                                        -*- Autoconf -*-
# Process this file with autoconf to produce a configure script.
AC_PREREQ(2.59)
#The next one is modified by sunq
#AC_INIT(FULL-PACKAGE-NAME,VERSION,BUG-REPORT-ADDRESS)
AC_INIT(hello,1.0)
# The next one is added by sunq
AM_INIT_AUTOMAKE(hello,1.0)
AC_CONFIG_SRCDIR([hello.c])
AC_CONFIG_HEADER([config.h])
# Checks for programs.
AC_PROG_CC
# Checks for libraries.
# Checks for header files.
# Checks for typedefs, structures, and compiler characteristics.
# Checks for library functions.
AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

下面对这个脚本文件进行解释：

（1）以“#”号开始的行为注释。

（2）AC_PREREQ 宏声明本文件要求的 autoconf 版本，如本例使用的版本是 2.59。

（3）AC_INIT 宏用来定义软件的名称和版本等信息，在本例中省略了 BUG-REPORT-ADDRESS，一般为作者的 E-mail。

（4）AM_INIT_AUTOMAKE 是笔者另加的，它是 automake 所必备的宏，也同前面一样，PACKAGE 是所要产生软件套件的名称，VERSION 是版本编号。

（5）AC_CONFIG_SRCDIR 宏用来侦测所指定的源码文件是否存在，以此来确定源码目录的有效性。在此处为当前目录下的 hello.c。

（6）AC_CONFIG_HEADER 宏用于生成 config.h 文件，以便 autoheader 使用。

（7）AC_CONFIG_FILES 宏用于生成相应的 Makefile 文件。

（8）中间的注释间可以分别添加用户测试程序、测试函数库、测试头文件等宏定义。

接下来首先运行 aclocal，生成一个“aclocal.m4”文件，该文件主要处理本地的宏定义，如下：

```
[root@localhost automake]# aclocal
```

接着运行 autoconf，生成“configure”可执行文件，如下：

```
[root@localhost automake]# autoconf
[root@localhost automake]# ls
```


华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

```
aclocal.m4  autom4te.cache  autoscan.log  configure  configure.in  hello.c
```

3．autoheader

接着使用 autoheader 命令，它负责生成 config.h.in 文件。该工具通常会从 acconfig.h 文件中复制用户附加的符号定义，因为此处没有附加符号定义，所以不需要创建 acconfig.h 文件，代码如下：

```
[root@localhost automake]# autoheader
```

4．automake

这是创建 Makefile 很重要的一步，automake 要用的脚本配置文件是 Makefile.am，用户需要自己创建相应的文件。然后，automake 工具转换成 Makefile.in。在该例中，笔者创建的文件为 Makefile.am，代码如下：

```
AUTOMAKE_OPTIONS=foreign
bin_PROGRAMS= hello
hello_SOURCES= hello.c
```

下面对该脚本文件的对应项进行解释。

（1）其中的 AUTOMAKE_OPTIONS 为设置 automake 的选项。由于 GNU（在第 1 章中已经有所介绍）对自己发布的软件有严格的规范，如必须附带许可证声明文件 COPYING 等，否则 automake 执行时会报错。automake 提供了 3 种软件等级：foreign、gnu 和 gnits，供用户选择，默认等级为 gnu。在本例中使用 foreign 等级，它只检测必需的文件。

（2）bin_PROGRAMS 定义要产生的执行文件名。如果要产生多个执行文件，每个文件名用空格隔开。

（3）hello_SOURCES 定义“hello”这个执行程序所需要的原始文件。如果“hello”这个程序是由多个原始文件所产生的，则必须把它所用到的所有原始文件都列出来，并用空格隔开。例如，若目标体“hello”需要“hello.c”、“sunq.c”、“hello.h”3 个依赖文件，则定义 hello_SOURCES=hello.c sunq.c hello.h。要注意的是，如果要定义多个执行文件，则对每个执行程序都要定义相应的 file_SOURCES。

接下来可以使用 automake 对其生成“configure.in”文件，在这里使用选项“--adding-missing”让 automake 自动添加一些必需的脚本文件，代码如下：

```
[root@localhost automake]# automake --add-missing
configure.in: installing './install-sh'
configure.in: installing './missing'
Makefile.am: installing 'depcomp'
[root@localhost automake]# ls
aclocal.m4     autoscan.log  configure.in  hello.c     Makefile.am  missing
autom4te.cache  configure     depcomp    install-sh  Makefile.in  config.h.in
```

可以看到，在 automake 之后就可以生成 configure.in 文件。

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

5．运行 configure

在这一步中，通过运行自动配置设置文件 configure，把 Makefile.in 变成了最终的 Makefile，如下：

```
[root@localhost automake]# ./configure
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build enVironment is sane... yes
checking for gawk... gawk
checking whether make sets $(MAKE)... yes
checking for Gcc... Gcc
checking for C compiler default output file name... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for suffix of executables...
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether Gcc accepts -g... yes
checking for Gcc option to accept ANSI C... none needed
checking for style of include used by make... GNU
checking dependency style of Gcc... Gcc3
configure: creating ./config.status
config.status: creating Makefile
config.status: executing depfiles commands
```

可以看到，在运行 configure 时收集了系统的信息，用户可以在 configure 命令中对其进行配置。在./configure 中的自定义参数有两种：一种是开关式（--enable-XXX 或--disable-XXX）；另一种是开放式，即后面要填入一串字符（--with-XXX=yyyy）参数。读者可以自行尝试其使用方法。另外，读者可以查看同一目录下的“config.log”文件，以方便调试。

到此为止，Makefile 就可以自动生成了。回忆整个步骤，用户不再需要定制不同的规则，而只需要输入简单的文件及目录名即可，这样就大大方便了用户的使用。autotools 生成 Makefile 的流程图如图 3.4 所示。

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

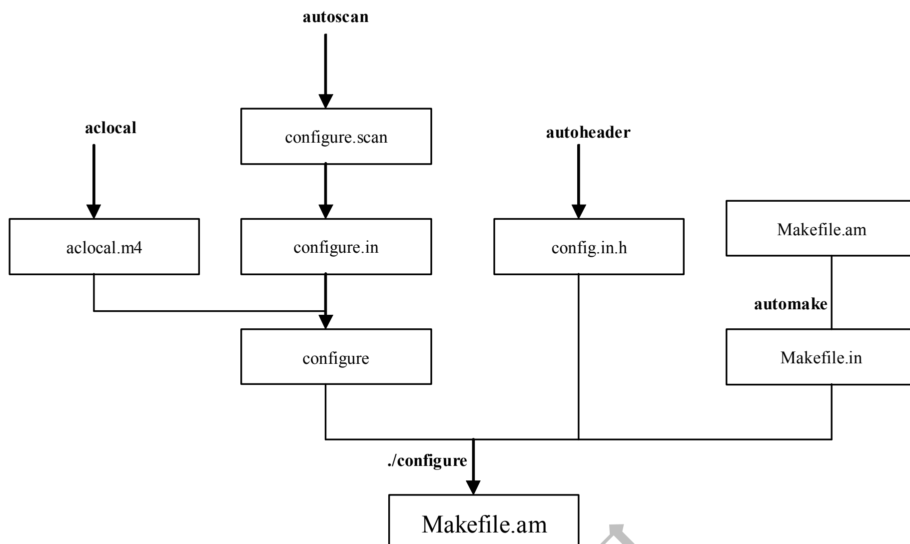


图 3.4　autotools 生成 Makefile 流程图

3.5.2　使用 autotools 生成的 Makefile

autotools 生成的 Makefile 除具有普通的编译功能外，还具有以下主要功能（感兴趣的读者可以查看这个简单的 hello.c 程序的 Makefile）。

1．make

输入 make 默认执行“make all”命令，即目标体为 all，其执行情况如下：

```
    [root@localhost automake]# make
    if    gcc    -DPACKAGE_NAME=\"\"    -DPACKAGE_TARNAME=\"\"    -DPACKAGE_VERSION=\"\"
-DPACKAGE_STRING=\"\" -DPACKAGE_BUGREPORT=\"\" -DPACKAGE=\"hello\" -DVERSION=\"1.0\"  -I.
-I.    -g -O2 -MT hello.o -MD -MP -MF ".deps/hello.Tpo" -c -o hello.o hello.c; \
    then mv -f ".deps/hello.Tpo" ".deps/hello.Po"; else rm -f ".deps/hello.Tpo"; exit
1; fi
    gcc  -g -O2   -o hello  hello.o
```

此时，在本目录下就生成了可执行文件“hello”，运行“./hello”能出现正常结果，代码如下：

```
    [root@localhost automake]# ./hello
    Hello!Autoconf!
```

2．make install

此时，会把该程序安装到系统目录中去，代码如下：

```
    [root@localhost automake]# make install
    if    gcc    -DPACKAGE_NAME=\"\"    -DPACKAGE_TARNAME=\"\"    -DPACKAGE_VERSION=\"\"
-DPACKAGE_STRING=\"\" -DPACKAGE_BUGREPORT=\"\" -DPACKAGE=\"hello\" -DVERSION=\"1.0\"  -I.
-I.    -g -O2 -MT hello.o -MD -MP -MF ".deps/hello.Tpo" -c -o hello.o hello.c; \
```

```
    then mv -f ".deps/hello.Tpo" ".deps/hello.Po"; else rm -f ".deps/hello.Tpo"; exit
1; fi
    gcc  -g -O2   -o hello  hello.o
    make[1]: Entering directory '/root/workplace/automake'
    test -z "/usr/local/bin" || mkdir -p -- "/usr/local/bin"
     /usr/bin/install -c 'hello' '/usr/local/bin/hello'
    make[1]: Nothing to be done for 'install-data-am'.
    make[1]: LeaVing directory '/root/workplace/automake'
```

此时，若直接运行 hello，也能出现正确结果，代码如下：

```
[root@localhost automake]# hello
Hello!Autoconf!
```

3．make clean

此时，make 会清除之前所编译的可执行文件及目标文件（object file, *.o），代码如下：

```
[root@localhost automake]# make clean
test -z "hello" || rm -f hello
rm -f *.o
```

4．make dist

此时，make 将程序和相关的文档打包为一个压缩文档以供发布，代码如下：

```
[root@localhost automake]# make dist
[root@localhost automake]# ls hello-1.0-tar.gz
hello-1.0-tar.gz
```

可见该命令生成了一个 hello-1.0-tar.gz 的压缩文件。

由上面的讲述读者不难看出，autotools 确实是软件维护与发布的必备工具，鉴于此，如今 GUN 的软件一般都是由 automake 来制作的。

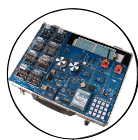
华清远见

3.6 本章习题



1．在 vi 中如何进入编辑模式？
2．在 vi 中如何删除整行？
3．gcc 的编译流程分为几个步骤？
4．使用 gcc 编译程序中，使用-g 选项的作用是什么？
5．制作一个简单的 Makefile 文件。

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com



华清远见
HQYJ.COM

第4章 存储管理

存储管理是操作系统的重要组成部分。Linux 操作系统采用了请求式分页虚拟存储管理方法。系统为每个进程提供了 4GB 的虚拟内存空间。各个进程的虚拟内存彼此独立。从计算机发展早期开始，就存在对大于系统中实际物理内存容量进行访问的需要。为了克服这种限制，系统开发者设计了许多策略，其中最成功的就是虚拟内存技术。本章主要讲解 Linux 系统的存储管理方式。

华清远见

华清远见
HQYJ.COM
华清远见教育集团官网：www.hqyj.com

4.1 进程虚存空间的管理



Linux 运行在 X86 架构时，进程的虚拟内存为 4GB。进程虚存空间的划分在系统初始化时由 GDT 确定，它定义在/arch/i386/kernel/head.S 文件中，代码如下：

```
        .quad 0x0000000000000000    /* NULL 描述符 */
        .quad 0x0000000000000000    /* 未使用*/
        .quad 0xc0c39a000000ffff    /* 内核代码段 1GB 在 0xc0000000 */
        .quad 0xc0c392000000ffff    /* 内核数据段 1GB 在 0xc0000000 */
        .quad 0x00cbfa000000ffff    /* 用户代码段 3GB 在 0x00000000 */
        .quad 0x00cbf2000000ffff    /* 用户数据段 3GB 在 0x00000000 */
        .quad 0x0000000000000000    /* 未使用 */
        .quad 0x0000000000000000    /* 未使用 */
        .fill 2*NR_TASKS,8,0        /* 各个进程 LDT 描述符和 TSS 描述符的空间 */
```

Linux 的存储管理主要是管理进程虚拟内存的用户区。进程虚拟内存的用户区分为代码段、数据段、堆栈，以及进程运行的环境变量、参数传递区域等。每个进程都用一个 mm_struct 结构体来定义它的虚存用户区。mm_struct 结构体首地址在任务结构体 task_struct 成员项 mm 中。

4.1.1 进程的虚存区域

虚存区域是虚存空间中一个连续的区域，在这个区域中的信息具有相同的操作和访问特性。每个虚拟区域用一个 vm_area_struct 结构体进行描述，它定义在/include/linux/mm_types.h 中，代码如下：

```
    /*
     * This struct defines a memory VMM memory area. There is one of these
     * per VM-area/task.  A VM area is any part of the process virtual memory
     * space that has a special rule for the page-fault handlers (ie a shared
     * library, the executable area etc).
     */
    struct vm_area_struct {
        struct mm_struct * vm_mm; /* The address space we belong to. */
        unsigned long vm_start;        /* Our start address within vm_mm. */
        /* The first byte after our end address within vm_mm. */
        unsigned long vm_end;
        /* linked list of VM areas per task, sorted by address */
        struct vm_area_struct *vm_next;

        pgprot_t vm_page_prot;         /* Access permissions of this VMA. */
        unsigned long vm_flags;        /* Flags, see mm.h. */

        struct rb_node vm_rb;
```


华清远见 HQYJ.COM 华清远见教育集团官网：www.hqyj.com

```
    /*
     * For areas with an address space and backing store,
     * linkage into the address_space->i_mmap prio tree, or
     * linkage to the list of like vmas hanging off its node, or
     * linkage of vma in the address_space->i_mmap_nonlinear list.
     */
    union {
        struct {
            struct list_head list;
            void *parent;/* aligns with prio_tree_node parent */
            struct vm_area_struct *head;
            } vm_set;

            struct raw_prio_tree_node prio_tree_node;
        } shared;

    /*
     * A file's MAP_PRIVATE vma can be in both i_mmap tree and anon_vma
     * list, after a COW of one of the file pages.  A MAP_SHARED vma
     * can only be in the i_mmap tree.  An anonymous MAP_PRIVATE, stack
     * or brk vma (with NULL file) can only be in an anon_vma list.
     */
    struct list_head anon_vma_node;    /* Serialized by anon_vma->lock */
    struct anon_vma *anon_vma;/* Serialized by page_table_lock */

    /* Function pointers to deal with this struct. */
    struct vm_operations_struct * vm_ops;

    /* Information about our backing store: */
    unsigned long vm_pgoff;        /* Offset (within vm_file) in PAGE_SIZE
                       units, *not* PAGE_CACHE_SIZE */
    struct file * vm_file;         /* File we map to (can be NULL). */
    void * vm_private_data;        /* was vm_pte (shared mem) */
    unsigned long vm_truncate_count;/* truncate_count or restart_addr */

#ifndef CONFIG_MMU
    atomic_t vm_usage;             /* refcount (VMAs shared if !MMU) */
#endif
#ifdef CONFIG_NUMA
    struct mempolicy *vm_policy;   /* NUMA policy for the VMA */
#endif
};
```

其中的各个域说明如下。

- vm_mm 指针指向进程的 mm_struct 结构体。
- vm_start 和 vm_end 虚拟区域的开始和终止地址。
- vm_flags 指出了虚存区域的操作特性。
  - VM_READ：虚存区域允许读取。
  - VM_WRITE：虚存区域允许写入。
  - VM_EXEC：虚存区域允许执行。

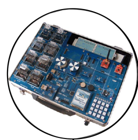
  - VM_SHARED ：虚存区域允许多个进程共享。
  - VM_GROWSDOWN：虚存区域可以向下延伸。
  - VM_GROWSUP：虚存区域可以向上延伸。
  - VM_SHM：虚存区域是共享存储器的一部分。
  - VM_LOCKED：虚存区域可以加锁。
  - VM_STACK_FLAGS：虚存区域作为堆栈使用。
- vm_page_prot 虚存区域的页面的保护特性。
- 若虚存区域映射的是磁盘文件或设备文件的内容，则 vm_inode 指向这个文件的 inode 结构体，否则 vm_inode 为 NULL。
- vm_offset 是该区域的内容相对于文件起始位置的偏移量，或相对于共享内存首址的偏移量。
- 所有 vm_area_struct 结构体链接成一个单向链表，vm_next 指向下一个 vm_area_struct 结构体。链表的首地址由 mm_struct 中成员项 mmap 指出。
- vm_ops 是指向 vm_operations_struct 结构体的指针。该结构体中包含指向各种操作的函数的指针。
- 所有 vm_area_struct 结构体组成一个 AVL 树（AVL 树是一种具有平衡结构的二叉树）。
  - vm_avl_left：左指针指向相邻的低地址虚存区域。
  - vm_avl_right：右指针指向相邻的高地址虚存区域。
  - mmap_avl：表示进程 AVL 树的根。
  - vm_avl_hight：表示 AVL 树的高度。
- vm_next_share 和 vm_prev_share，把有关的 vm_area_struct 结合成一个共享内存时使用的双向链表。

华清远见

4.1.2　虚存空间的映射和虚存区域的建立

虚拟内存（虚存）使计算机可以操纵更大的地址空间，还可以使系统中的每一个进程都有自己的虚拟地址空间。这些虚拟的地址空间是相互完全分离的，所以，运行一个应用程序的进程不会影响另外的进程。另外，硬件的虚拟内存机制允许对内存区写保护。这可以防止代码和数据被错误的程序覆盖。内存映射可以将 CPU 的虚拟地址空间映射到系统的物理内存。

核心的共享虚拟内存机制，虽然允许进程拥有分离（虚拟）的地址空间，但有时也需要进程之间共享内存。例如，系统中可能有多个进程运行命令解释程序 bash 。尽管可以在每一个进程的虚拟地址空间都拥有一份 bash 的备份文件，但更好的方法是在物理内存中只拥有一份备份文件，所有运行 bash 的进程共享代码。动态链接库是多个进程共享执行代码的另一个常见例子。另外，共享内存也经常用于进程间通信机制，两个



或多个进程可以通过共同拥有的内存交换信息。Linux 系统支持系统 V 的共享内存 IPC 机制。

Linux 虚存采用动态地址映射方式，即进程的地址空间和存储空间的对应关系是在程序的执行过程中实现的。进程使用的是虚拟地址，因此，它对每个地址的访问都需通过 MMU 把虚拟地址转化为内存的物理地址。

动态地址映射使 Linux 可以实现进程在主存中的动态重定位、虚存段的动态扩展和移动，也为虚存的实现提供了基础。当 Linux 中的进程映像执行时，需要调入可执行映像的内容。但并不用把这些数据直接调入物理内存，而只需要把这些数据放入该进程的虚拟内存区。只有当执行需要这些数据时才真正调入内存。这种进程的映像和虚拟进程空间的连接称为内存映像。当需要将进程映像调入进程的虚拟内存空间时，需要申请一段合适的虚拟内存空间，这时需要用到 mmap 系统调用来获得所需的内存空间。

Linux 使用 do_mmap()函数完成可执行映像向虚存区域的映射，由它建立有关的虚存区域 do_mmap()函数定义在 include\linux\mm.h 文件中：

```
static inline unsigned long do_mmap(struct file *file, unsigned long addr,
    unsigned long len, unsigned long prot,
    unsigned long flag, unsigned long offset)
{
    unsigned long ret = -EINVAL;
    if ((offset + PAGE_ALIGN(len)) < offset)
        goto out;
    if (!(offset & ~PAGE_MASK))
        ret = do_mmap_pgoff(file, addr, len, prot, flag, offset >> PAGE_SHIFT);
out:
    return ret;
}
```

- addr 是虚存区域在虚拟内存空间的起始地址。
- len 是这个虚存区域的长度。
- file 是指向该文件结构体的指针，若 file 为 NULL，则称为匿名映射（Anonymous Mapping）。
- offset 是相对于文件起始位置的偏移量。
- prot 指定了虚存区域的访问特性。
  - PROT_READ　0x1：对虚存区域允许读取。
  - PROT_WEITE　0x2：对虚存区域允许写入。
  - PROT_EXEC　0x4：虚存区域（代码）允许执行。
  - PROT_NONE　0x0：不允许访问该虚存区域。
- flag 指定了虚存区域的属性。
  - MAP_FIXED：指定虚存区域固定在 addr 的位置上。
  - MAP_SHARED：指定对虚存区域的操作是作用在共享页面上。
  - MAP_PRIVATE 指定了对虚存区域的写入操作将引起页面复制。

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

4.2 内存空间/地址类型



CPU 地址空间（Address Space）是一段表示内存位置的地址范围。在 X86 架构下，地址空间有 3 种：物理地址空间、线性地址空间和逻辑地址空间（虚拟地址空间）。而在 ARM 体系结构下只有物理地址空间和逻辑地址空间（虚拟地址空间）。

物理地址是一个系统中可用的真实的硬件地址。例如，一个系统有 128MB 内存，它的合法地址范围是 0~0x8000000（以十六进制表示）。每个地址都对应于所安装的 SIMMs 中的一组晶体管，而且对应于处理器地址总线上的一组特定信号。

逻辑地址则是 CPU 所能处理的地址空间的总和，对于 32 位 CPU 而言，它的逻辑地址空间是 4G.B。采用逻辑地址空间的好处是每个用户进程都有自己独立的运行空间，而不用管自己在物理内存的实际位置。在 Linux 系统中，4GB 的地址空间由 Linux 内核与 Linux 应用程序共同分享，如图 4.1 所示。

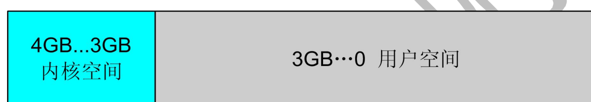


图 4.1 内核空间与用户空间

用户空间使用 0x00000000~0xBFFFFFFF 共 3GB 的地址空间，用户态进程可以直接访问此空间。内核空间则使用 0xC0000000~0xFFFFFFFF 剩下的 1GB 地址空间，存放内核访问的代码和数据，用户态进程不能直接访问。用户进程只有通过中断或系统调用进入核心态时才有权利访问。

在逻辑地址和物理地址之间相互转换的工作是 CPU 的内存管理单元（MMU）完成的。Linux 内核负责告诉 MMU 如何把逻辑页面映射到物理页面。通常，内核需要维护每个进程的逻辑地址和物理地址对照表，在切换进程时，更新 MMU 的对照表信息。而 MMU 在进程提出内存请求时会自动完成实际的地址转换工作。

在 X86 体系结构上，把线性地址映射到物理地址分为两个步骤，整个过程如图 4.2 所示。提供给进程的线性地址被分为 3 个部分：一个页目录索引、一个页表索引和一个偏移量。页目录（Page Directory）是一个指向页表的指针数组，页表（Page Table）是一个指向页面的指针数组，因此，地址映射就是一个跟踪指针链的过程。一个页目录能够确定一个页表，继而得到一个页面，然后页面中的偏移量（Offset）能够指出该页面中的一个地址。

为了进行更详细而准确的描述，给定页目录索引中的页目录项保存着存储在物理内存中的一个页表地址，给定页表索引中的页表项保存着物理内存上相应物理页面的基地

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

址，然后线性地址的偏移量加到这个物理地址上形成最终物理页面内的目的地址。MMU 对线性地址的转换如图 4.2 所示。

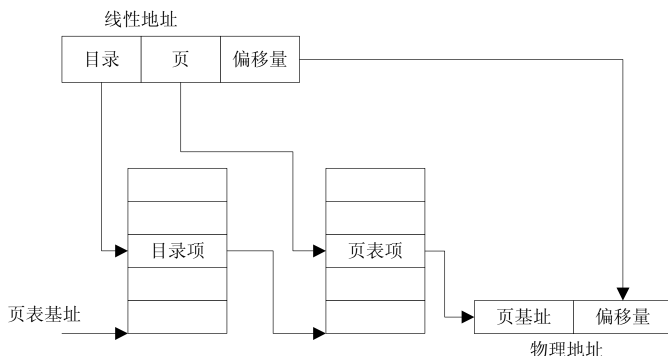


图 4.2　MMU 线性地址到物理地址转换图

4.3 分页机制与 MMU



Linux 的内存管理采用页式管理，使用多级页表，动态地址转换机构与主存、辅存共同实现虚拟内存。每个用户进程拥有 4GB 的虚拟地址空间，进程在运行过程中可以动态浮动和扩展，为用户提供了透明的、灵活有效的内存使用方式。这正是进程被分配一个逻辑地址空间的原因之一。即使每个进程有相同的逻辑地址空间，通过分页机制，相应进程的物理地址也都是不同的，因此，它们在物理上不会彼此重叠。

从内核的角度来看，逻辑和物理地址都被划分成固定大小的页面。每个合法的逻辑页面恰好处于一个物理页面中，方便 MMU 的地址转换。当地址转换无法完成时（例如，由于给定的逻辑地址不合法或由于逻辑页面没有对应的物理页面），MMU 将产生中断，向核心发出信号。Linux 核心可以处理这种页面错误（Page Fault）问题。

MMU 也负责增强内存保护，当一个应用程序试图在它的内存中对一个已标明是只读的页面进行写操作时，MMU 也会产生中断错误，通知内核。在没有 MMU 的情况下，内核不能防止一个进程非法存取其他进程的内存空间。

每个进程都有一套自己的页目录与页表，其中页目录的基地址是关键，通过它才能查到逻辑所对应的物理地址。页目录的基地址是每个进程的私有资源，保存在该进程的 task_struct 对象的 mm_struct 结构变量 mm 中。

mm_struct 结构在 include\linux\mm_types.h 中定义，task_struct 结构定义将在第 5 章进行介绍。

```
struct mm_struct {
    struct vm_area_struct * mmap;          /*list of VMAs */
    struct rb_root mm_rb;
```

```
    struct vm_area_struct * mmap_cache;     /* last find_vma result */
    unsigned long (*get_unmapped_area) (struct file *filp,
                unsigned long addr, unsigned long len,
                unsigned long pgoff, unsigned long flags);
    void (*unmap_area) (struct mm_struct *mm, unsigned long addr);
    unsigned long mmap_base;        /* base of mmap area */
    unsigned long task_size;        /* size of task vm space */
    unsigned long cached_hole_size;
                /* if non-zero, the largest hole below free_area_cache */
    unsigned long free_area_cache;
                    /*first hole of size cached_hole_size or larger */
    pgd_t * pgd;
    atomic_t mm_users;              /* How many users with user space? */
    atomic_t mm_count;
       /* How many references to "struct mm_struct" (users count as 1) */
    int map_count;                  /* number of VMAs */
    struct rw_semaphore mmap_sem;
    spinlock_t page_table_lock;
                        /* Protects page tables and some counters */

    struct list_head mmlist;
    mm_counter_t _file_rss;
    mm_counter_t _anon_rss;

    unsigned long hiwater_rss;/* High-watermark of RSS usage */
    unsigned long hiwater_vm; /* High-water virtual memory usage */

    unsigned long total_vm, locked_vm, shared_vm, exec_vm;
    unsigned long stack_vm, reserved_vm, def_flags, nr_ptes;
    unsigned long start_code, end_code, start_data, end_data;
    unsigned long start_brk, brk, start_stack;
    unsigned long arg_start, arg_end, env_start, env_end;

    unsigned long saved_auxv[AT_VECTOR_SIZE]; /* for /proc/PID/auxv */

    cpumask_t cpu_vm_mask;

    /* Architecture-specific MM context */
    mm_context_t context;
    unsigned int faultstamp;
    unsigned int token_priority;
    unsigned int last_interval;

    unsigned long flags; /* Must use atomic bitops to access the bits */

    struct core_state *core_state; /* coredumping support */

    /* aio bits */
    rwlock_t     ioctx_list_lock;  /* aio lock */
    struct kioctx     *ioctx_list;
#ifdef CONFIG_MM_OWNER
    struct task_struct *owner;
#endif
```

```
#ifdef CONFIG_PROC_FS
    /* store ref to file /proc/<pid>/exe symlink points to */
    struct file *exe_file;
    unsigned long num_exe_file_vmas;
#endif
#ifdef CONFIG_MMU_NOTIFIER
    struct mmu_notifier_mm *mmu_notifier_mm;
#endif
};
```

在进程切换时，CPU 会把新进程的页目录基地址填入 CPU 的页目录寄存器，供 MMU 使用。当新进程有地址访问时，MMU 会根据被访问地址的最高 10 位从页目录中找到对应的页表基地址，然后根据次高 10 位再从页表中找到对应的物理地址的页首，最后根据剩下的 12 位偏移量与页首地址找到逻辑地址对应的真正物理地址。在 Linux 内核中，有关页目录与页表的操作已被定义成一套宏，使用起来非常方便。相关定义都在<asm/pgtable.h>文件中。

```
#define pgd_index(addr)         ((addr) >> PGDIR_SHIFT)
#define __pgd_offset(addr) pgd_index(addr)
```

提取被访问地址中用于在页目录中索引的部分。

```
#define pgd_offset(mm, addr)   ((mm)->pgd+pgd_index(addr))
```

根据逻辑地址及页目录基地址找到对应的页目录项。

```
#define pte_none(pte)       (!pte_val(pte))
```

判断页表项是否为空。

```
#define pte_clear(ptep)         set_pte((ptep), __pte(0))
```

将对应页表项清为空。

```
#define pte_page(x)         (virt_to_page(__va(pte_val((x)))))
```

找出页表项对应的页首地址。

```
#define mk_pte(page,pgprot)
({
    pte_t __pte;
    pte_val(__pte) = __pa(page_address(page)) + pgprot_val(pgprot);       \
    __pte;
})
```

根据页首地址和访问属性合成为一个合法的页表项。

```
static inline pte_t pte_modify(pte_t pte, pgprot_t newprot)
{
    pte_val(pte) = (pte_val(pte) & _PAGE_CHG_MASK) | pgprot_val(newprot);
    return pte;
}
```

用新的属性替代页表项中旧的属性。

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

4.4 高速缓存



Linux 使用了许多与高速缓存相关的内存管理策略。

1．缓冲区高速缓存

缓冲区高速缓存中包含被块设备驱动使用的数据缓冲，这些缓冲单元的大小一般都固定（如 512 B），包含从块设备读出或写入的信息块。块设备是仅能够以固定大小块进行读/写操作的设备，所有的硬盘都是块设备。利用设备标识符和所需块号作索引可以在缓冲区高速缓存中迅速找到数据块。块设备只能够通过缓冲区高速缓存来存取。如果数据在缓冲区缓存中可以找到，则无须从物理块设备（如硬盘）中读取，这样可以加速设备的访问速度。

2．页高速缓存

页高速缓存用来加速块设备上可执行映像文件与数据文件的存取。它每次缓冲一个页面的文件内容。页面从块设备上读入内存后放入页高速缓存中。

3．交换高速缓存

只有修改过的页面才存储在交换文件中。如果这些页面在写入到交换文件后没有被修改，则下次被交换出内存时就不必再进行更新写操作。这些页面都可以丢弃在交换频繁发生的系统中。交换高速缓存可以减少很多不必要且耗时的块设备操作。

4．硬件高速缓存

常见的硬件高速缓存是处理器中的指令和数据 Cache，它缓存 CPU 最近访问过的指令和数据，使 CPU 不需要到内存中获取数据。Cache 是 CPU 与内存之间的桥梁。目前，常见的 CPU 中 Cache 的实现按读/写方式分类，不外乎如下几类。

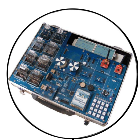
1）贯穿读出式（Look Through）

该方式将 Cache 隔在 CPU 与主存之间，CPU 将主存的所有数据请求都首先送到 Cache，由 Cache 自行在自身查找。如果命中，则切断 CPU 对主存的请求，并将数据送出；如果没有命中，则将数据请求传给主存。该方法的优点是降低了 CPU 对主存的请求次数，缺点是延迟了 CPU 对主存的访问时间。

2）旁路读出式（Look Aside）

在这种方式中，CPU 发出数据请求时，并不是单通道地穿过 Cache，而是向 Cache 和主存同时发出请求。由于 Cache 速度更快，如果命中，则 Cache 在将数据回送给 CPU 的同时，还来得及中断 CPU 对主存的请求；如果没有命中，则 Cache 不做任何动作，由 CPU 直接访问主存。

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com



它的优点是没有时间延迟，缺点是每次 CPU 对主存的访问都存在，这样，就占用了一部分总线时间。

3）写穿式（Write Through）

任一从 CPU 发出的写信号送到 Cache 的同时，也写入主存，以保证主存的数据能同步更新。它的优点是操作简单，但由于主存的速度慢，从而降低了系统的写速度并占用了总线的时间。

4）回写式（Copy Back）

为了克服贯穿读出式中每次数据写入时都要访问主存，从而导致系统写速度降低并占用总线时间的弊病，尽量减少对主存的访问次数，又有了回写式。它的工作原理为：数据一般只写到 Cache，这样有可能出现 Cache 中的数据得到更新而主存中的数据不变（数据陈旧）的情况。但此时可在 Cache 中设定一个标识地址及数据陈旧的信息，只有当 Cache 中的数据被再次更改时，才将原更新的数据写入主存相应的单元中，然后再接受再次更新的数据。这样保证了 Cache 和主存中的数据不产生冲突。

4.5 内存区域 Zone



Linux 通过伙伴算法管理和分配页，但由于硬件的原因，内存中的不同区域会有不同的特性。主要有以下两个问题：

- 一些硬件只能用某些特定的内存地址来执行 DMA。
- 一些体系结构中有一些内存不能永久映射到内核空间上。

因此某些内存必须从特定区域上分配，不能由单一的伙伴系统管理。为了区分这些内存区域，Linux 使用了 3 个 Zone，每个 Zone 由一个自己的伙伴系统来管理，如下：

- ZONE_DMA 包含可以用来执行 DMA 操作的内存。
- ZONE_NORMAL 包含可以正常映射到虚拟地址的内存区域。
- ZONE_HIGHMEM 包含不能永久映射到内核地址空间的内存区域。

这些区域的划分和具体的体系结构有关，例如，某些体系结构中 ZONE_NORMAL 覆盖了所有的内存区域，而另外两个区均为空。在 X86 系统中，ZONE_DMA 为 0~16MB 的内存范围，ZONE_HIGHMEM 包含了所有高于 896MB 的物理内存，ZONE_NORMAL 覆盖其余部分。这个规定可在<linux/mmzone.h>中找到，源代码如下：

```
#ifdef CONFIG_ZONE_DMA
     ZONE_DMA,
#endif
#ifdef CONFIG_ZONE_DMA32
     /*
      * x86_64 needs two ZONE_DMAs because it supports devices that are
```


华清远见 HQYJ.COM

华清远见教育集团官网：www.hqyj.com

```
     * only able to do DMA to the lower 16M but also 32 bit devices that
     * can only do DMA areas below 4G.
     */
    ZONE_DMA32,
#endif
    /*
     * Normal addressable memory is in ZONE_NORMAL. DMA operations can be
     * performed on pages in ZONE_NORMAL if the DMA devices support
     * transfers to all addressable memory.
     */
    ZONE_NORMAL,
#ifdef CONFIG_HIGHMEM

    ZONE_HIGHMEM,
#endif
    ZONE_MOVABLE,
    __MAX_NR_ZONES
};
```

这些内存区域被分别管理，而在分配时，不同的任务会从不同的区域分配，例如，DMA 任务只能从 ZONE_DMA 中分配内存，而普通的内存请求则会依次尝试从 ZONE_NORMAL、 ZONE_HIGHMEM 和 ZONE_DMA 中分配。在分配器看来，这 3 个区只是 3 个不同的内存池对象，用相同的方法即可进行处理。

zone 结构表示区在<linux/mmzone.h>文件中定义，它包含了该伙伴系统的所有信息。zone 结构如下：

```
struct zone {
    /* Fields commonly accessed by the page allocator */
    unsigned long     pages_min, pages_low, pages_high;
    unsigned long     lowmem_reserve[MAX_NR_ZONES];

#ifdef CONFIG_NUMA
    int node;
    /*
     * zone reclaim becomes active if more unmapped pages exist.
     */
    unsigned long     min_unmapped_pages;
    unsigned long     min_slab_pages;
    struct per_cpu_pageset     *pageset[NR_CPUS];
#else
    struct per_cpu_pageset     pageset[NR_CPUS];
#endif
    /*
     * free areas of different sizes
     */
    spinlock_t         lock;
#ifdef CONFIG_MEMORY_HOTPLUG
    /* see spanned/present_pages for more description */
    seqlock_t          span_seqlock;
#endif
    struct free_area  free_area[MAX_ORDER];
```


华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com




```
#ifndef CONFIG_SPARSEMEM
    /*
     * Flags for a pageblock_nr_pages block. See pageblock-flags.h.
     * In SPARSEMEM, this map is stored in struct mem_section
     */
    unsigned long     *pageblock_flags;
#endif /* CONFIG_SPARSEMEM */


    ZONE_PADDING(_pad1_)

    /* Fields commonly accessed by the page reclaim scanner */
    spinlock_t        lru_lock;
    struct {
        struct list_head list;
        unsigned long nr_scan;
    } lru[NR_LRU_LISTS];
    unsigned long     recent_rotated[2];
    unsigned long     recent_scanned[2];

    unsigned long     pages_scanned;      /* since last reclaim */
    unsigned long     flags;          /* zone flags, see below */

    /* Zone statistics */
    atomic_long_t     vm_stat[NR_VM_ZONE_STAT_ITEMS];
    int prev_priority;

    /*
     * The target ratio of ACTIVE_ANON to INACTIVE_ANON pages on
     * this zone's LRU.  Maintained by the pageout code.
     */
    unsigned int inactive_ratio;
…
    ZONE_PADDING(_pad2_)
    /* Rarely used or read-mostly fields */

    wait_queue_head_t * wait_table;
    unsigned long     wait_table_hash_nr_entries;
    unsigned long     wait_table_bits;

    /*
     * Discontig memory support fields.
     */
    struct pglist_data    *zone_pgdat;
    /* zone_start_pfn == zone_start_paddr >> PAGE_SHIFT */
    unsigned long     zone_start_pfn;
…
    unsigned longspanned_pages;    /* total size, including holes */
    unsigned longpresent_pages;    /* amount of memory (excluding holes) */

    /*
     * rarely used fields:
     */
```


华清远见 HQYJ.COM

华清远见教育集团官网：www.hqyj.com

```
	const char		*name;
} --cacheline_internodealigned_in_smp;
```

其中，zone_mem_map 成员用来指向所有这个 zone 管理到 page 数组集合。

4.6 获得内存页面



系统中对于物理页有大量的需求，当程序映像加载到内存中时，操作系统需要分配页；当程序结束执行并卸载时，操作系统需要释放这些页。另外，为了存放相关的数据结构（如页表自身），也需要物理页。这种用于分配和回收页的机制和数据结构对于维护虚拟内存子系统的效率是非常重要的。

系统中的所有物理页都使用 page 数据结构来描述。每一个物理页对应一个 page 变量。一个 zone 的所有 page 变量集合形成数组，由 zone 的 zone_mem_map 成员指针指向数组的起始地址，页数组的初始化在系统启动时完成。

页分配器的算法是在伙伴系统之上的，伙伴系统将内存区域组织为以页为单位的块，*n* 称为该块的“级别”，具有相同级别的块用链表接在一起。每次分配必须指定一个级别，并以该级别块的大小为单位。

在分配时，依次从级别 *n* 到最大级别开始搜索，直到找到一个非空的块为止。如果这个非空块级别不是 *n*，则将其拆成两份，一份放到其对应的级别的空闲块中，另一份如果还不是级别 *n* 就继续拆分，直到最后返回那个级别为 *n* 的块。

在回收时，首先计算出被回收的块的伙伴，然后查看这个伙伴是否在级别为 *n* 的空闲链中。如果找到了，则将这个块与这个伙伴合并（伙伴从空闲链删除，并修整“当前块”的位置即可），然后 $n := n+1$，继续这个合并过程。当伙伴不在该空闲链中时，合并过程结束。free_area 所管理的内存如图 4.3 所示。

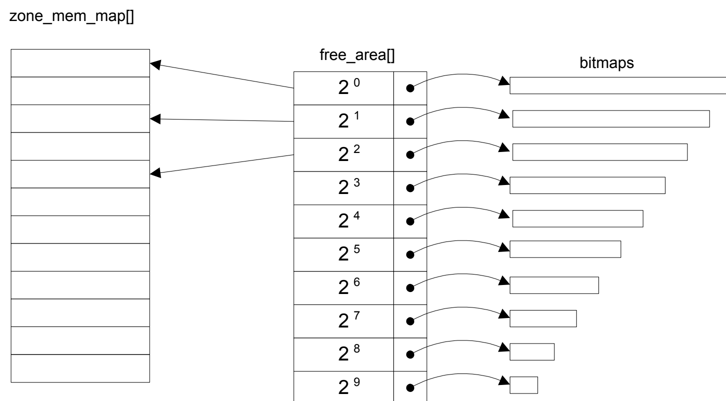


图 4.3　free_area 管理内存结构图

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

上面的算法可以非常有效地分配和回收内存，但同时也限制了分配的灵活性（页面数必须是 2 的指数）。zone 中与这个算法相关的 field 主要是：

```
struct free_area    free_area[MAX_ORDER];
```

这里就是各个级别的空闲链的入口。free_area 结构是一个非常简单的结构，其中包括一个链表项和该级别的链表元素数目：

```
struct free_area {
    struct list_head    free_list;
    unsigned long       nr_free;
};
```

而这个 free_list 所指向的对象则是 struct page 中的一个链表成员 lru，通过它将 page 串在一起组成该级别的空闲链。free_list->next 和 free_list->prev 分别指向空闲链的头和尾。当级别 $n > 0$ 时，这个块含有一个以上的 page，而这时只使用块中的首个 page 来进行链接，其他（连续相邻的）page 则隐含地存在于这个块中，数据结构如图 4.4 所示（图中的链接实际上都是双链，一组小括号表示一个 page 结构）。

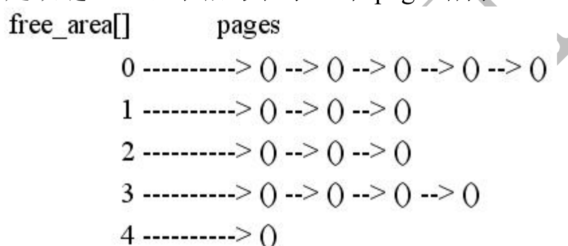


图 4.4　数据结构

伙伴系统的数据结构由 mm/page_alloc.c 中的各个函数来建立和维护。主要包括下面几个函数：

```
struct page *__alloc_pages_internal(gfp_t gfp_mask, unsigned int order,
            struct zonelist *zonelist, nodemask_t *nodemask)
{
    const gfp_t wait = gfp_mask & __GFP_WAIT;
    enum zone_type high_zoneidx = gfp_zone(gfp_mask);
    struct zoneref *z;
    struct zone *zone;
    struct page *page;
    struct reclaim_state reclaim_state;
    struct task_struct *p = current;
    int do_retry;
    int alloc_flags;
    unsigned long did_some_progress;
    unsigned long pages_reclaimed = 0;

    might_sleep_if(wait);

    if (should_fail_alloc_page(gfp_mask, order))
        return NULL;
```


华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

```
restart:
    z = zonelist->_zonerefs;  /* the list of zones suitable for gfp_mask */

    if (unlikely(!z->zone)) {
        /*
         * Happens if we have an empty zonelist as a result of
         * GFP_THISNODE being used on a memoryless node
         */
        return NULL;
    }

    page = get_page_from_freelist(gfp_mask|__GFP_HARDWALL, nodemask, order,
            zonelist, high_zoneidx, ALLOC_WMARK_LOW|ALLOC_CPUSET);
    if (page)
        goto got_pg;

    if (NUMA_BUILD && (gfp_mask & GFP_THISNODE) == GFP_THISNODE)
        goto nopage;

    for_each_zone_zonelist(zone, z, zonelist, high_zoneidx)
        wakeup_kswapd(zone, order);
    alloc_flags = ALLOC_WMARK_MIN;
    if ((unlikely(rt_task(p)) && !in_interrupt()) || !wait)
        alloc_flags |= ALLOC_HARDER;
    if (gfp_mask & __GFP_HIGH)
        alloc_flags |= ALLOC_HIGH;
    if (wait)
        alloc_flags |= ALLOC_CPUSET;
    page = get_page_from_freelist(gfp_mask, nodemask, order, zonelist,
                        high_zoneidx, alloc_flags);
    if (page)
        goto got_pg;

    /* This allocation should allow future memory freeing. */

rebalance:
    if (((p->flags & PF_MEMALLOC) || unlikely(test_thread_flag(TIF_MEMDIE)))
            && !in_interrupt()) {
        if (!(gfp_mask & __GFP_NOMEMALLOC)) {
nofail_alloc:
            /* go through the zonelist yet again, ignoring mins */
            page = get_page_from_freelist(gfp_mask, nodemask, order,
                zonelist, high_zoneidx, ALLOC_NO_WATERMARKS);
            if (page)
                goto got_pg;
            if (gfp_mask & __GFP_NOFAIL) {
                congestion_wait(WRITE, HZ/50);
                goto nofail_alloc;
            }
        }
        goto nopage;
    }
```




```
    /* Atomic allocations - we can't balance anything */
    if (!wait)
        goto nopage;

    cond_resched();

    /* We now go into synchronous reclaim */
    cpuset_memory_pressure_bump();
    /*
     * The task's cpuset might have expanded its set of allowable nodes
     */
    cpuset_update_task_memory_state();
    p->flags |= PF_MEMALLOC;
    reclaim_state.reclaimed_slab = 0;
    p->reclaim_state = &reclaim_state;

    did_some_progress = try_to_free_pages(zonelist, order, gfp_mask);

    p->reclaim_state = NULL;
    p->flags &= ~PF_MEMALLOC;

    cond_resched();

    if (order != 0)
        drain_all_pages();

    if (likely(did_some_progress)) {
        page = get_page_from_freelist(gfp_mask, nodemask, order,
                    zonelist, high_zoneidx, alloc_flags);
        if (page)
            goto got_pg;
    } else if ((gfp_mask & __GFP_FS) && !(gfp_mask & __GFP_NORETRY)) {
        if (!try_set_zone_oom(zonelist, gfp_mask)) {
            schedule_timeout_uninterruptible(1);
            goto restart;
        }

        page = get_page_from_freelist(gfp_mask|__GFP_HARDWALL, nodemask,
            order, zonelist, high_zoneidx,
            ALLOC_WMARK_HIGH|ALLOC_CPUSET);
        if (page) {
            clear_zonelist_oom(zonelist, gfp_mask);
            goto got_pg;
        }

        /* The OOM killer will not help higher order allocs so fail */
        if (order > PAGE_ALLOC_COSTLY_ORDER) {
            clear_zonelist_oom(zonelist, gfp_mask);
            goto nopage;
        }

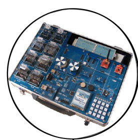
        out_of_memory(zonelist, gfp_mask, order);
```

```
            clear_zonelist_oom(zonelist, gfp_mask);
            goto restart;
        }
        pages_reclaimed += did_some_progress;
        do_retry = 0;
        if (!(gfp_mask & __GFP_NORETRY)) {
            if (order <= PAGE_ALLOC_COSTLY_ORDER) {
                do_retry = 1;
            } else {
                if (gfp_mask & __GFP_REPEAT &&
                    pages_reclaimed < (1 << order))
                        do_retry = 1;
            }
            if (gfp_mask & __GFP_NOFAIL)
                do_retry = 1;
        }
        if (do_retry) {
            congestion_wait(WRITE, HZ/50);
            goto rebalance;
        }

    nopage:
        if (!(gfp_mask & __GFP_NOWARN) && printk_ratelimit()) {
            printk(KERN_WARNING "%s: page allocation failure."
                " order:%d, mode:0x%x\n",
                p->comm, order, gfp_mask);
            dump_stack();
            show_mem();
        }
    got_pg:
        return page;
    }
```

该函数负责根据需求从 zonelist 所指的 zone 中分配 page，分配过程中会首先严守每个区的保留底线（各个区都会试图保留一定量的最低空闲页面数以备急用）进行分配。如果失败，则会试图唤醒 kswapd 守护进程进行内存交换，以空出更多内存，同时尝试以宽松一些的方式（放宽各个 zone 的保留底线）再次尝试；如果依然失败，则视当时的上下文而定。如果上下文允许做一些调整工作，则尝试由此释放一些内存，如果成功了，则再次尝试分配或者跳转到最初的分配代码再次尝试。如果不允许或调整之后，最后用无底线方式再次尝试。

```
    void __free_pages(struct page *page, unsigned int order)
    {
        if (put_page_testzero(page)) {
            if (order == 0)
                free_hot_page(page);
            else
                __free_pages_ok(page, order);
        }
    }
```



这个函数负责释放已经分配的页面，它对于 order == 0 的页面调用 free_hot_page 函数，一般的请求用__free_page_ok 函数来处理。

4.7 slab 分配器



Linux 内核中有许多内存动态分配的需求，而其中的对象大小也参差不齐，Linux 内核提供了 slab 层，扮演了通用数据结构缓存层的角色。 slab 层根据对象的类型来分组不同的 Cache，每个 Cache 存放不同类型的对象，例如，一个 Cache 被用来存储 task_struct，而另一个存放 Inode 等。这些 Cache 包含几个 slab，而 slab 由一个或多个物理上连续的 page 组成。对于一般的数据结构，每个 slab 只有一个页即可。

每个 slab 都包含一些对象成员，即被管理的数据结构。系统分配对象时就从 slab 中取得。首先从这个 Cache 中部分满的 slab 中分配，如果没有这样的 slab，便从空的 slab 中分配，如果也没有，就创建一个新的 slab 来分配即可。

因为每个 slab 是包含同一种对象的 Cache 块，它对对象的分配和释放会变得更为容易和快速。另外，由于每个对象在释放时几乎处于分配好并且初始化好的状态，还可以节省不少初始化的时间。例如，分配 inode 变量，首先需要一块 malloc(sizeof(inode)) 大小的内存，然后初始化 Inode 中的数据成员，在使用完毕后用 free 释放分配的内存。实际上，在 free 之后内存中的内容和刚刚初始化时差不多，例如，Inode 的引用计数 Count 一定是降为零。

Kernel 有许多数据结构都是还原为初始化时的状态后才会 free 掉，例如，一个 Mutex lock 初始化时和释放时都处于 Unlock 状态。因此，只要在 Cache 初始化时，就将对象置于合法状态，以后每次分配对象的这些 field 一定是确定的，从而不必重复初始化，可以节省不少开销。Linux 的 kmem_cache_create 中有一个参数 ctor 初始化函数，可以被用做这一目的，但 Linux 似乎并没有使用 slab 的这一特性（因为它没用调用 ctor 函数）。

每个 Cache 都用 struct kmem_cache_s 表示，其中有一个重要的成员变量——struct kmem_list3 lists。

这个结构体中存放了该 cache 中空、部分满、满的 slab 的队列：

```
struct kmem_list3 {
    struct list_head slabs_partial;
    struct list_head slabs_full;
    struct list_head slabs_free;
    unsigned long free_objects;
    int free_touched;
    unsigned long next_reap;
    struct array_cache *shared;
};
```

其中的 3 个 list 均指向一个 struct slab 的列表。struct slab 的定义如下：

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com
华清远见

```
struct slab {
    struct list_head    list;
    unsigned long       colouroff;
    void                *s_mem;              // slab 中的第一个对象
    unsigned int        inuse;
    kmem_bufctl_t       free;                // 第一个空闲对象(如果有的话)
};
```

slab 描述符（struct slab 对象）本身也要占用内存。它们可以放在 slab 自身开始的地方，或在 slab 之外另行分配。slab 分配器创建新的 slab 正是通过上面的页分配器进行的。

Cache 的创建和销毁是非常重要的内容，Cache 的创建通过下面这个函数完成：

```
kmem_cache_t *kmem_cache_create (const char *name, size_t size, size_t align,
        unsigned long flags, void (*ctor)(void*, kmem_cache_t *, unsigned long),
        void (*dtor)(void*, kmem_cache_t *, unsigned long))
```

其中，第一个参数表示 Cache 的名字，第二个参数是对象的大小（而不是 Cache 的大小，后者会自动调整），第三个参数是高速缓存内第一个对象的偏移。flags 是可选的设置项，用来控制 Cache 的行为。最后两个参数 ctor 和 dtor 分别是 Cache 的构造器和解构器，分别在新的 page 添加到 Cache 中，以及从 Cache 中删除时调用。如果用不上，传入 NULL 即可。

kmem_cache_create 函数在成功时返回所创建的 Cache 的指针，失败了则返回 NULL。其该函数主要根据 size 来计算出 slab size 以及其描述符 struct slab 是否内联，并且初始化其数据结构等。有趣的一点是，kmem_cache_t 本身也是从一个 cache_cache 通过 kmem_alloc 函数来分配的，不过这个 Cache 的初始化应该是另外完成的。

```
int kmem_cache_destroy (kmem_cache_t * cachep)
```

该函数用来销毁给定的高速缓存。这个函数通常在模块的注销代码中被调用。调用这个函数之前必须确保存在以下两个条件：

- Cache 中的所有 slab 都必须为空（即所有对象都已经被收回）。
- 在调用 kmem_cache_destroy 期间不能再访问这个 Cache，调用者必须保证这种同步。

对象的分配和释放可以使用 kmem_cache_alloc 函数实现：

```
void * kmem_cache_alloc (kmem_cache_t *cachep, int flags)
```

该函数从给定的 cachep 中返回一个指向对象的指针。在必要时（没有空闲对象时），会调用更低层的 kmem_get_pages()来获取新的页，并初始化（调用 ctor）等。与该函数对应的释放函数如下：

```
void kmem_cache_free (kmem_cache_t *cachep, void *objp)
```

下面给出一个使用实例。在 Linux 内核的各个模块中都使用了 slab 提供的内存分配方案，它的使用可以大大节省内存开销，同时提高运行效率。例如，在 driver/usb/uhci.c 文件中：

```
static int __init uhci_hcd_init(void)
```


华清远见 HQYJ.COM 华清远见教育集团官网：www.hqyj.com




```
{
	int retval = -ENOMEM;

	if (usb_disabled())
		return -ENODEV;

	printk(KERN_INFO "uhci_hcd: " DRIVER_DESC "%s\n",
			ignore_oc ? ", overcurrent ignored" : "");
	set_bit(USB_UHCI_LOADED, &usb_hcds_loaded);

	if (DEBUG_CONFIGURED) {
		errbuf = kmalloc(ERRBUF_LEN, GFP_KERNEL);
		if (!errbuf)
			goto errbuf_failed;
		uhci_debugfs_root = debugfs_create_dir("uhci", NULL);
		if (!uhci_debugfs_root)
			goto debug_failed;
	}

	uhci_up_cachep = kmem_cache_create("uhci_urb_priv",
		sizeof(struct urb_priv), 0, 0, NULL);
	if (!uhci_up_cachep)
		goto up_failed;

	retval = pci_register_driver(&uhci_pci_driver);
	if (retval)
		goto init_failed;

	return 0;

init_failed:
	kmem_cache_destroy(uhci_up_cachep);

up_failed:
	debugfs_remove(uhci_debugfs_root);

debug_failed:
	kfree(errbuf);

errbuf_failed:

	clear_bit(USB_UHCI_LOADED, &usb_hcds_loaded);
	return retval;
}
```

初始化程序首先分配出一个 urb_priv 类型的 Cache 实体，这样在后面需要为 urb_priv 类型的变量分配内存时，就可以直接从 Cache 实体中分配出来了。从 Cache 中分配内存的代码也在本文件中。

```
static struct urb_priv *uhci_alloc_urb_priv(struct uhci *uhci, struct urb *urb)
{
	struct urb_priv *urbp;
```


华清远见 HQYJ.COM 华清远见教育集团官网：www.hqyj.com

```
        urbp = kmem_Cache_alloc(uhci_up_Cachep, in_interrupt() ? SLAB_ATOMIC :
SLAB_KERNEL);
        if (!urbp) {
            err("uhci_alloc_urb_priv: couldn't allocate memory for urb_priv\n");
            return NULL;
        }

        memset((void *)urbp, 0, sizeof(*urbp));

        urbp->inserttime = jiffies;
        urbp->fsbrtime = jiffies;
        urbp->urb = urb;
        urbp->dev = urb->dev;

        INIT_LIST_HEAD(&urbp->td_list);
        INIT_LIST_HEAD(&urbp->queue_list);
        INIT_LIST_HEAD(&urbp->complete_list);

        urb->hcpriv = urbp;

        if (urb->dev != uhci->rh.dev) {
            if (urb->transfer_buffer_length) {
                urbp->transfer_buffer_dma_handle = pci_map_single(uhci->dev,
                    urb->transfer_buffer, urb->transfer_buffer_length,
                    usb_pipein(urb->pipe) ? PCI_DMA_FROMDEVICE :
                    PCI_DMA_TODEVICE);
                if (!urbp->transfer_buffer_dma_handle)
                    return NULL;
            }

            if (usb_pipetype(urb->pipe) == PIPE_CONTROL && urb->setup_packet) {
                urbp->setup_packet_dma_handle = pci_map_single(uhci->dev,
                    urb->setup_packet, sizeof(devrequest),
                    PCI_DMA_TODEVICE);
                if (!urbp->setup_packet_dma_handle)
                    return NULL;
            }
        }

        return urbp;
    }
```

4.8 kmalloc



核心的 kmalloc 内存分配函数和应用层的 malloc 函数很相似，只是这个函数运行得很快——除非它被阻塞。kmalloc 函数不清零它获得的内存空间，分配给它的区域仍存

放着原有的数据。kmalloc 函数在<linux/slab.h>中定义，它是通过下一层的__kmalloc 完成内存分配的。

```
static void *kmalloc(size_t size, int flags)
void * __kmalloc (size_t size, int flags)
```

kmalloc 函数的第一个参数是 size（大小），第二个参数 flags 是优先权参数，它会使 kmalloc 函数在寻找空闲页较困难时改变函数的行为。最常用的优先权是 GFP_KERNEL，它的意思是该内存分配（内部是通过调用 get_free_pages 来实现的，所以名字中带 GFP）是由运行在内核态的进程调用的。也就是说，调用它的函数属于某个进程，使用 GFP_KERNEL 优先权允许 kmalloc 函数在系统空闲内存低于水平线 min_free_pages 时延迟分配函数的返回。当空闲内存太少时，kmalloc 函数会使当前进程进入睡眠，等待空闲页的出现。

新的页面可以通过以下几种途径获得。一种方法是换出其他页：因为对换需要时间，进程会等待它完成，这时内核可以调度执行其他的任务。因此，每个调用 kmalloc(GFP_KERNEL)的内核函数都应该是可重入的。

并非使用 GFP_KERNEL 优先权后一定正确，有时 kmalloc 是在进程上下文之外调用的。例如，在中断处理、任务队列处理和内核定时器处理时发生。这些情况下，current 进程就不能进入睡眠，这时就应该使用优先权 GFP_ATOMIC。原子性（Atomic）的内存分配允许使用内存的空闲位，而与 min_free_pages 值无关。实际上，这个最低水平线值的存在就是为了能满足原子性的请求。但由于内核并不允许通过换出数据或缩减文件系统缓冲区来满足这种分配请求，所以必须还有一些真正可以获得的空闲内存。

kmalloc 还定义了其他一些优先权，但都不经常使用，其中一些只在内部的内存管理算法中使用。另一个值的注意的优先权是 GFP_NFS，它会使得 NFS 文件系统缩减空闲列表到 min_free_pages 值以下。

除了这些常用的优先权，kmalloc 还可以识别一个位域——GFP_DMA。GFP_DMA 标识位要和 GFP_KERNEL 与 GFP_ATOMIC 优先权一起使用来分配用于直接内存访问（DMA）的内存页。

系统物理内存的管理是由内核负责的，物理内存只能按页大小进行分配。这就需要一个面向页的分配技术以取得计算机内存管理上最大的灵活性。类似 malloc 函数的简单的线性的分配技术不再有效了。在像 Linux 内核这样的面向页的系统中，如果内存使用线性分配的策略就很难维护。空洞的处理很快就会成为一个问题，会导致内存浪费，降低系统的性能。

Linux 是通过维护页面池来处理 kmalloc 的分配要求的，这样，页面就可以很容易地放进或取出页面池。为了能够满足超过 PAGE_SIZE 字节数大小的内存分配请求，mm/slab.c 文件维护页面簇的列表。每个页面簇都存放着连续若干页，可用于 DMA 分配。Linux 所使用的分配策略的最终方案是，内核只能分配一些预定义的固定大小的字节数组。如果申请任意大小的内存空间，那么很可能系统会多分配一点。

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

这些预定义的内存大小一般“稍小于 2 的某次方”（而在更新的实现中系统管理的内存大小恰好为 2 的某次方）。如果能记住这一点，就可以更有效地使用内存。例如，需要一个 2000B 左右的缓冲区，最好还是申请 2000B，而不要申请 2048B。申请恰好是 2 的幂次的内存空间是最糟糕的情况了——内核会分配两倍于申请空间大小的内存。

4.9 高端内存



一般情况下，Linux 在初始化时总是尽可能地将所有的物理内存映射到内核地址空间中去。如果内核地址空间起始于 0xC0000000，为 vmalloc 保留的虚拟地址空间是 128MB，那么最多只能有（1GB-128MB）的物理内存直接映射到内核空间中，内核可以直接访问。如果还有更多的物理内存，就称为高端内存，内核不能直接访问，只能通过修改页表映射后才能进行访问。

内存分区可以使内核页分配更加合理。当系统物理内存大于 1GB 时，内核不能将所有的物理内存都预先映射到内核空间中，这样就产生了高端内存，高端内存最适于映射到用户进程空间中。预映射的部分可直接用于内核缓冲区，其中有一小块可用于 DMA 操作的内存，留给 DMA 操作分配用，一般不会轻易分配。内存分区还可以适应不连续的物理内存分布，是非一致性内存存取体系（NUMA）的基础。

在 32 位机器上，页表通常只可以存储在低端内存中。低端内存只限于物理内存的前 896 MB，同时还要满足内核其余的大部分要求。在应用程序使用了大量进程并映射了大量内存的情况下，低端内存可能很快就不够用了。在 2.6 内核中有一个配置选项称为 Highmem PTE，让页表条目可以存放在高端内存中，释放出更多的低端内存区域给那些必须放在这里的其他内核数据结构，使用这些页表条目的进程会稍微慢一些。不过，对于那些有大量进程在运行的系统来说，将页表存储到高端内存中可以在低端内存区域挤出更多的内存。

华清远见

4.10 虚拟内存的申请和释放



申请和释放较小且连续的内存空间时，使用 kmalloc()和 kfree()函数在物理内存中进行分配；申请较大的内存空间时，可以使用 vmalloc()函数。由 vmalloc()函数申请的内存空间在虚拟内存中是连续的，它们映射到物理内存时，可以使用不连续的物理页面，而且仅把当前访问的部分放在物理页面中。本节要介绍的内存分配函数是 vmalloc。尽管这段区域在物理上可能是不连续的（要访问其中的每个页面都必须独立地调用函数 __get_free_page），内核却认为它们在地址上是连续的。分配的内存空间被映射到内核数

据段中，用户空间是不可见的，这一点与其他分配技术不同。vmalloc 发生错误时返回 0（NULL 地址），成功时返回一个指向一个大小为 size 的线性地址空间的指针。vmalloc 函数在核心中所分配的内存由 vm_struct 结构的链表所支持，如图 4.5 所示。

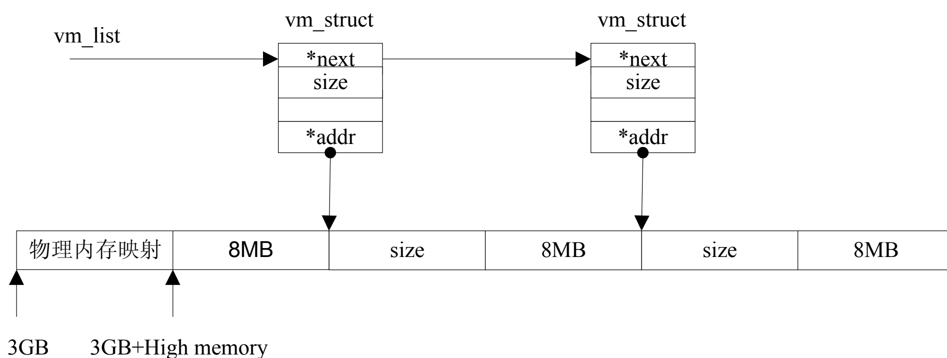


图 4.5 vm_struct 链表所管理的虚存空间

该函数及其相关函数的原型如下：

```
    void* vmalloc(unsigned long size);
    void vfree(void* addr);
    void *ioremap(unsigned long phys_addr, size_t size, unsigned long flags, unsigned
long align);
```

使用 vmalloc 时应将<linux/vmalloc.h>包含进来。与其他内存分配函数不同的是，vmalloc 返回很“高”的地址值——这些地址要高于物理内存的顶部。由于 vmalloc 对页表调整后允许用连续的“高”地址访问分配得到的页面，因此，处理器是可以访问返回得到的内存区域的。内核能和其他地址一样使用 vmalloc 返回的地址，但程序中用到的这个地址与地址总线上的地址并不相同。

用 vmalloc 分配得到的地址是不能在微处理器之外使用的，因为它们只有在处理器的分页单元上才有意义。当驱动程序需要真正的物理地址时（像外设用于驱动系统总线的 DMA 地址），这样的地址是不能通过 vmalloc 函数分配的。正确使用 vmalloc 函数的场合是为软件分配一大块连续的用于缓冲的内存区域。注意，vmalloc 的开销要比 __get_free_pages 大，因为它处理获取内存还要建立页表。因此，不值得用 vmalloc 函数只分配一页的内存空间。

vmalloc 分配的内核虚拟内存与 kmalloc/__get_free_pages 分配的内核逻辑内存位于不同的区间，不会重叠。因为内核空间被分区管理，各司其职。用户空间被分配在 0~3GB 之间，3GB 之后紧跟着是物理内存映射区间，然后才是 vmalloc_start 开始的用于 vmalloc 分配内存的地址空间，如图 4.6 所示。

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

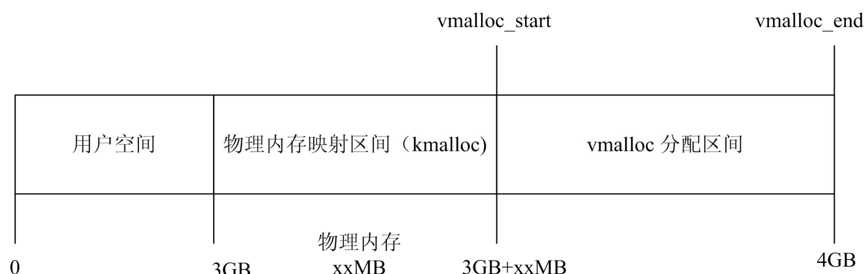


图 4.6　内存分配区间示意图

使用 vmalloc 函数的一个例子函数是 create_module 系统调用，它利用 vmalloc 函数来获取被创建模块需要的内存空间。而在 insmod 调用重定位模块代码后，将会调用 memcpy_fromfs 函数把模块本身复制进分配而得的空间内。

用 vmalloc 分配得到的内存空间用 vfree 函数来释放，这就像是要用 kfree 函数来释放 kmalloc 函数分配得到的内存空间。

和 vmalloc 一样，ioremap 也建立新的页表，但和 vmalloc 不同的是，ioremap 实际上并不分配内存。ioremap 的返回值是一个虚拟地址，可以用来访问指定的物理内存区域，得到的这个虚拟地址最后要调用 vfree 来释放掉。

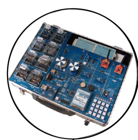
ioremap 用于将高内存空间的 PCI 缓冲区映射到用户空间。例如，如果 VGA 设备的帧缓冲区被映射到地址 0xf0000000（典型的一个值）后，ioremap 就可以建立正确的页表，从而让处理器可以访问。而系统初始化时建立的页表只是用于访问低于物理地址空间的内存区域。系统的初始化过程并不检测 PCI 缓冲区，而是由各个驱动程序自己负责管理自己的缓冲区。

如果希望驱动程序能在不同的平台间移植，那么使用 ioremap 时就要小心。在一些平台上是不能直接将 PCI 内存区域映射到处理器的地址空间的，如在 Alpha 上就不行。此时就不能像普通内存区域那样对重映射区域进行访问，而应该用 readb 函数或其他一些 I/O 函数。这些函数可以在不同平台间移植。

对 vmalloc 和 ioremap 函数可分配的内存空间大小并没有什么限制，但为了能检测到程序员犯的一些错误，vmalloc 不允许分配超过物理内存大小的内存空间。但是，vmalloc 函数请求过多的内存空间会产生一些和调用 kmalloc 函数时相同的问题。

ioremap 和 vmalloc 函数都是面向页的（它们都会修改页表），因此，分配或释放的内存空间实际上都会上调为最近的一个页边界。而且，ioremap 函数并不考虑如何重映射不是页边界的物理地址。

华清远见
HQYJ.COM
华清远见教育集团官网：www.hqyj.com



4.11 本章习题



1. 什么是逻辑地址空间？
2. 什么是物理地址空间？
3. 简述逻辑地址和物理地址之间的映射？
4. Linux 采用的是哪种存储管理方法？
5. 在内核中如何申请和释放内存？

华清远见
HQYJ.COM
华清远见

第5章 操作系统进程

在传统的操作系统中，为了提高系统资源的利用率，通常采用多道程序技术，将多个程序同时载入内存，并使之并发执行。此时，作为资源分配和独立运行的基本单位都是进程。操作系统所具有的几大特征，也都是围绕进程形成的。在操作系统中，进程是最重要的概念。本章将讲解进程的相关内容。

华清远见
HQYJ.COM
华清远见教育集团官网：www.hqyj.com



华清远见



5.1 进程的基本概念



在计算机使用过程中，我们经常谈及的概念是程序。作为最终用户，我们关心系统中哪些程序在运行，需要关闭哪个程序。但是从操作系统的范畴来说，我们使用更多的是进程。

进程和程序虽然有一定的联系，但是绝不能混为一谈。在传统的操作系统中，程序并不能独立运行，作为资源分配和独立运行的基本单元都是进程。程序是一个普通文件，是机器代码指令和数据的集合，这些指令和数据存储在磁盘上的一个可执行映像（Executable Image）中。进程是由正文段（Text）、用户数据段（User Segment）及系统数据段（System Segment）共同组成的一个执行环境，它是一个动态实体。程序是硬盘上存放的一个文件（代码）。当程序运行时，它也就成为了进程。进程的组成部分如下。

- 正文段：存放被执行的机器指令。这个段是只读的，它允许系统中正在运行的两个或多个进程之间能够共享这一代码，但是不能对其内容进行更改。
- 用户数据段：存放进程在执行时直接进行操作的所有数据，包括进程使用的全部变量在内。显然，这里包含的信息可以被改变。虽然进程之间可以共享正文段，但是每个进程需要有它自己的专用用户数据段。
- 系统数据段：该段有效地存放程序运行的环境。这也是进程和程序不同的一个原因之一。

如图 5.1 所示为程序和进程的区别。

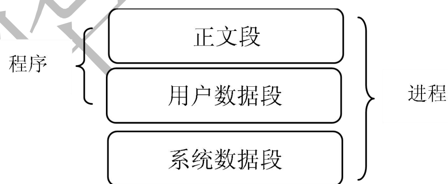


图 5.1　程序和进程的区别

Windows 和 Linux 操作系统都属于多任务系统，可以同时运行多个进程。我们经常使用的 Windows 任务管理器可以清楚地列出当前系统运行的进程，如图 5.2 所示，在图中可以看到，该系统此时正在运行的进程有 29 个。

华清远见

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

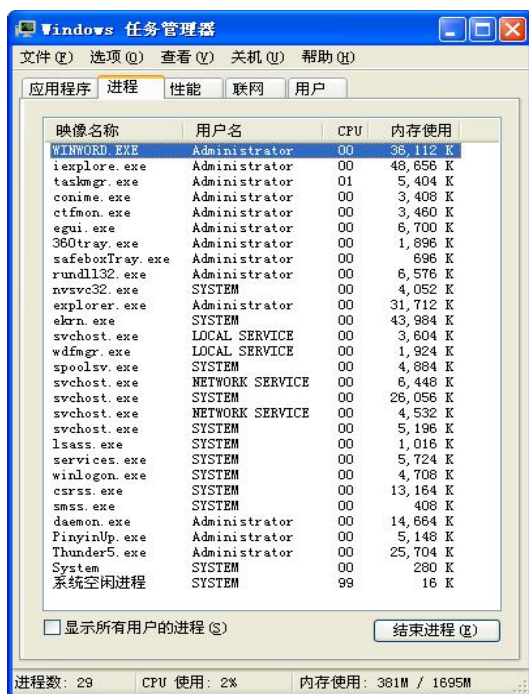


图 5.2　Windows 任务管理器

在谈到进程时，还要涉及线程的概念。线程是系统分配处理器时间资源的基本单元，或者说进程之内独立执行的一个单元。对于操作系统来说，其调度单元是线程。一个进程至少包括一个线程，通常将该线程称为主线程。一个进程从主线程的执行开始进而创建一个或多个附加线程，就是所谓基于多线程的多任务。

在 Linux 2.6 内核中，Linux 采用了更为先进的线程模型：NPTL（Native POSIX Thread Library）。与传统的 LinuxThreads 线程模型相比，NPTL 与 POSIX 标准兼容，并且性能提升明显，也具备更好的可伸缩性。对 Linux 内核而言，线程和进程没有本质的区别，它们都是调度的基本单位（实际上，Linux 内核基于进程机制实现线程），本书重点介绍进程的内容。

5.2　Linux 系统进程



5.2.1　Linux 进程基础

Linux 进程一般分为交互进程、批处理进程和守护进程 3 类。与 Windows 任务管理器一样，在 Linux 中可以通过 ps 命令查看系统当前的进程，例如，下面的命令将列出系

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

统所有的进程：

```
[root@localhost ~]# ps -aux
```

如果进程太多，可以把 ps 命令的输出保存到一个文件中：

```
[root@localhost ~]# ps -aux  > mypsout
```

ps 是 Linux 进程管理中最重要的一个命令，它提供了很多选项参数，如表 5.1 所示。

表 5.1　ps 命令的选项参数

参　数	功　能
l	长格式输出
u	按用户名和启动时间的顺序来显示进程
j	用任务格式来显示进程
f	用树形格式来显示进程
a	显示所有用户的所有进程（包括其他用户）
x	显示无控制终端的进程
r	显示运行中的进程
ww	避免详细参数被截断

常用的选项组合是 aux，使用不同的选项会产生不同的输出结果。表 5.2 列出了使用 ps 命令的输出列说明。

表 5.2　ps 的输出列说明

列	说　明
USER	进程的属主
PID	进程的 ID
PPID	父进程
%CPU	进程占用的 CPU 百分比
%MEM	占用内存的百分比
NI	进程的 NICE 值，数值大，表示较少占用 CPU 时间
VSZ	进程虚拟大小
RSS	驻留中页的数量
WCHAN	正在等待的进程资源
TTY	终端 ID
STAT	进程状态
START	启动进程的时间
TIME	进程消耗 CPU 的时间
COMMAND	命令的名称和参数

华清远见

表 5.2 中的 STAT 列表示进程的状态。在 Linux 操作系统中，系统有几种不同的状态，在 CPU 的调度下，进行状态的切换。表 5.3 列出了 Linux 系统中的各种进程状态。

表 5.3　Linux 进程状态

值	说　明
D	该进程处于睡眠状态
R	该进程处于运行状态
S	该进程处于睡眠状态
T	该进程处于停止或被追踪状态
W	进入内存交换
X	死掉的进程
Z	僵尸进程
<	优先级高的进程
N	优先级较低的进程
L	有些页被锁进内存
s	进程的领导者
l	多线程进程
+	位于后台的进程组

5.2.2　进程描述符

Linux 系统的每一个可调度实体都有一个进程描述符。进程描述符可以表示进程的各种状态信息，是内核操作进程的手段。进程描述符用 task_struct 数据结构表示，该结构包含一个进程所拥有的各种信息，非常庞大，在内核文件的 sched.h 中定义。下面给出 task_struct 数据结构的片断。

```
struct task_struct {
    volatile long state;  /* -1 unrunnable, 0 runnable, >0 stopped */
    void *stack;
    atomic_t usage;
    unsigned int flags;   /* per process flags, defined below */
    unsigned int ptrace;
    int lock_depth;       /* BKL lock depth */
    int prio, static_prio, normal_prio;
    unsigned int rt_priority;
    const struct sched_class *sched_class;
    struct sched_entity se;
    struct sched_rt_entity rt;
    unsigned char fpu_counter;
    s8 oomkilladj; /* OOM kill score adjustment (bit shift). */
    unsigned int policy;
    cpumask_t cpus_allowed;
    struct list_head tasks;
```


华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com




```
	struct mm_struct *mm, *active_mm;
/* task state */
	struct linux_binfmt *binfmt;
	int exit_state;
	int exit_code, exit_signal;
	int pdeath_signal;  /*  The signal sent when the parent dies  */

	unsigned int personality;
	unsigned did_exec:1;
	pid_t pid;
	pid_t tgid;
	struct task_struct *real_parent; /* real parent process */
	struct task_struct *parent; /* recipient of SIGCHLD, wait4() reports */
	struct list_head children;/* list of my children */
	struct list_head sibling; /* linkage in my parent's children list */
	struct task_struct *group_leader;  /* threadgroup leader */
	struct list_head ptraced;
	struct list_head ptrace_entry;
	/* PID/PID hash table linkage. */
	struct pid_link pids[PIDTYPE_MAX];
	struct list_head thread_group;
	struct completion *vfork_done;		/* for vfork() */
	int __user *set_child_tid;		/* CLONE_CHILD_SETTID */
	int __user *clear_child_tid;		/* CLONE_CHILD_CLEARTID */
	cputime_t utime, stime, utimescaled, stimescaled;
	cputime_t gtime;
	cputime_t prev_utime, prev_stime;
	unsigned long nvcsw, nivcsw; /* context switch counts */
	struct timespec start_time; 		/* monotonic time */
	struct timespec real_start_time;	/* boot based time */
	unsigned long min_flt, maj_flt;
	struct task_cputime cputime_expires;
	struct list_head cpu_timers[3];

/* process credentials */
	uid_t uid,euid,suid,fsuid;
	gid_t gid,egid,sgid,fsgid;
	struct group_info *group_info;
	kernel_cap_t   cap_effective, cap_inheritable, cap_permitted, cap_bset;
	struct user_struct *user;
	unsigned securebits;
/* file system info */
	int link_count, total_link_count;
#ifdef CONFIG_SYSVIPC
/* ipc stuff */
	struct sysv_sem sysvsem;
#endif
#ifdef CONFIG_DETECT_SOFTLOCKUP
/* hung task detection */
	unsigned long last_switch_timestamp;
	unsigned long last_switch_count;
#endif
/* CPU-specific state of this task */
```


华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

```
        struct thread_struct thread;
    /* filesystem information */
        struct fs_struct *fs;
    /* open file information */
        struct files_struct *files;
    /* namespaces */
        struct nsproxy *nsproxy;
    /* signal handlers */
        struct signal_struct *signal;
        struct sighand_struct *sighand;

        sigset_t blocked, real_blocked;
        sigset_t saved_sigmask;    /* restored if set_restore_sigmask() was used */
        struct sigpending pending;
        unsigned long sas_ss_sp;
        size_t sas_ss_size;
        int (*notifier)(void *priv);
        void *notifier_data;
        sigset_t *notifier_mask;
        struct audit_context *audit_context;
        seccomp_t seccomp;
    /* Thread group tracking */
        u32 parent_exec_id;
        u32 self_exec_id;
    /* Protection of (de-)allocation: mm, files, fs, tty, keyrings */
        spinlock_t alloc_lock;
        /* Protection of the PI data structures: */
        spinlock_t pi_lock;
    /* journalling filesystem info */
        void *journal_info;
    /* stacked block device info */
        struct bio *bio_list, **bio_tail;
        atomic_t fs_excl; /* holding fs exclusive resources */
        struct rcu_head rcu;
        unsigned long timer_slack_ns;
        unsigned long default_timer_slack_ns;

        struct list_head  *scm_work_list;
    };
```

在这个结构中，对一个进程进行了全面描述。例如，每一个进程都有自己的进程号，该数字在系统中是唯一的，它存放在进程描述符的成员变量 pid 中：

```
    pid_t pid;
```

同样，线程组号存放在成员变量 tgid 中：

```
    pid_t tpid;
```

这个结构大致可以分为以下几类：

- 进程状态。记录进程是在等待、运行还是死锁。
- 调度信息。由哪个调度函数调度，怎样调度等。
- 进程的通信状况。



- 因为要插入进程树，所以必须有联系父子兄弟的指针。
- 时间信息。如计算好执行的时间，以便 CPU 分配。
- 标号。决定改进程归属。
- 可以读/写打开的一些文件信息。
- 进程上下文和内核上下文。
- 处理器上下文。
- 内存信息。

5.2.3 进程的状态与转换

进程对系统资源的竞争和进程之间的并发执行，使得一个进程在从创建到销毁的这个生命周期内要经历各种不同状态的变化。一个进程的各种状态在 task_struct 结构中通过成员变量 state、exit_state 记录。那么内核究竟为进程定义了几种状态？我们可以通过查看 sched.h 文件中定义的宏查找到答案。

```
#define TASK_RUNNING        0
#define TASK_INTERRUPTIBLE 1
#define TASK_UNINTERRUPTIBLE    2
#define __TASK_STOPPED      4
#define __TASK_TRACED 8
/* in tsk->exit_state */
#define EXIT_ZOMBIE         16
#define EXIT_DEAD       32
/* in tsk->state again */
#define TASK_DEAD       64
#define TASK_WAKEKILL 128
```

其中各个状态的含义如下。

- TASK_RUNNING：表示进程正在运行，或者是进程获得了除处理器以外的全部资源，一旦获得处理器，即可运行。
- TASK_INTERRUPTIBLE：表示进程处于阻塞态，处于该状态的进程睡眠在相应资源的等待队列上，当该类资源再次有效时，系统会唤醒进程，并转换到运行态（TASK_RUNNING）。
- TASK_UNINTERRUPTIBLE：与 TASK_INTERRUPTIBLE 态相似，不同的是，该状态下的进程一直等待，直到所需要的资源有效或等待超时从而由系统唤醒。
- TASK_STOPPED：表示进程处于暂停状态，通过任务控制信号 SIGSTOP 可以将 TASK_RUNNING 状态的进程转换到此状态；在此状态下的进程收到 SIGCONT 信号后会转换到 TASK_RUNNING 状态。
- TASK_TRACED：表示该进程处于监控状态，用于程序的 BUG 调试。
- EXIT_ZOMBIE：表示进程处于僵尸状态，在这个状态下的进程只能转换到 EXIT_DEAD 状态。僵尸状态的产生是由于父进程死亡而被终止的进程，但是

华清远见 HQYJ.COM 华清远见教育集团官网：www.hqyj.com

在 task 数据中仍然保留 task_struct 结构。

- EXIT_DEAD：表示父进程已经获得了该进程的记账信息，该进程可以被销毁。
- TASK_WAKEKILL：表示该进程已经退出且不需要父进程来回收。

提示

阻塞操作是指在执行设备操作时，如果没有获得资源，则进程挂起，直到满足可操作的条件再进行操作。非阻塞操作的进程在不能进行设备操作时，并不挂起。

在 Linux 的 2.6.25 内核中，引入了一种新的进程状态：TASK_KILLABLE，它用于将进程置为睡眠状态，可以替代有效但可能无法终止的 TASK_UNINTERRUPTIBLE 进程状态，以及易于唤醒但更加安全的 TASK_INTERRUPTIBLE 进程状态。但是在更高版本的内核中，Linux 又引入了 TASK_WAKEKILL 状态，随着这个状态的出现，内核又定义了几个便于设置状态的宏：

```
/* Convenience macros for the sake of set_task_state */
#define TASK_KILLABLE (TASK_WAKEKILL | TASK_UNINTERRUPTIBLE)
#define TASK_STOPPED  (TASK_WAKEKILL | __TASK_STOPPED)
#define TASK_TRACED   (TASK_WAKEKILL | __TASK_TRACED)

/* Convenience macros for the sake of wake_up */
#define TASK_NORMAL   (TASK_INTERRUPTIBLE | TASK_UNINTERRUPTIBLE)
#define TASK_ALL      (TASK_NORMAL | __TASK_STOPPED | __TASK_TRACED)

/* get_task_state() */
#define TASK_REPORT   (TASK_RUNNING | TASK_INTERRUPTIBLE | \
                  TASK_UNINTERRUPTIBLE | __TASK_STOPPED | \
                  __TASK_TRACED)
```

通过上面的代码，读者应该会更清楚 TASK_STOPPED 和__TASK_STOPPED 的区别了。Linux 系统通过一系列机制，每个进程的状态不断转换，从而实现多任务同时进行，其状态转换图如图 5.3 所示。

获得 CPU 而正在运行的进程若申请不到某个资源，则调用 sleep_on() 或 interruptible_sleep_on()睡眠，其 task_struct 挂到相应的等待队列。如果调用 sleep_on()睡眠，则其状态变为 TASK_UNINTERRUPTIBLE。如果调用 interruptible_sleep_on()睡眠，则其状态变为 TASK_INTERRUPTIBLE。sleep_on() 或 interruptible_sleep_on() 将调用 schedule()函数把睡眠进程释放的 CPU 分配给 run-queue 队列的某个就绪进程。

状态为 TASK_INTERRUPTIBLE 的睡眠进程当申请的资源有效时被唤醒（如 wake_up_interruptible()），也可以由信号（signal）或定时中断唤醒。而状态为 TASK_UNINTERRUPTIBLE 的睡眠进程只有当它申请的资源有效时才能被唤醒（如 wake_up()），不能被信号（Signal）、定时中断唤醒。唤醒后，进程状态改为 TASK_RUNNING，并进入运行队列。

进程执行系统调用 sys_exit()或收到 SIG_KILL 信号而调用 do_exit()时，进程状态变为 TASK_ZOMBIE，释放所申请的资源，同时启动 schedule()把 CPU 分配给运行队列中

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

其他就绪进程。若进程通过系统调用设置 PF_SYSTRACE，则在系统调用返回前，进入 ptrace()，状态变为 TASK_STOPPED，CPU 分配给运行队列中其他就绪进程。只有通过其他进程发送 SIG_KILL 或 SIG_CONT，才能把 TASK_STOPPED 进程唤醒，重新进入运行队列。

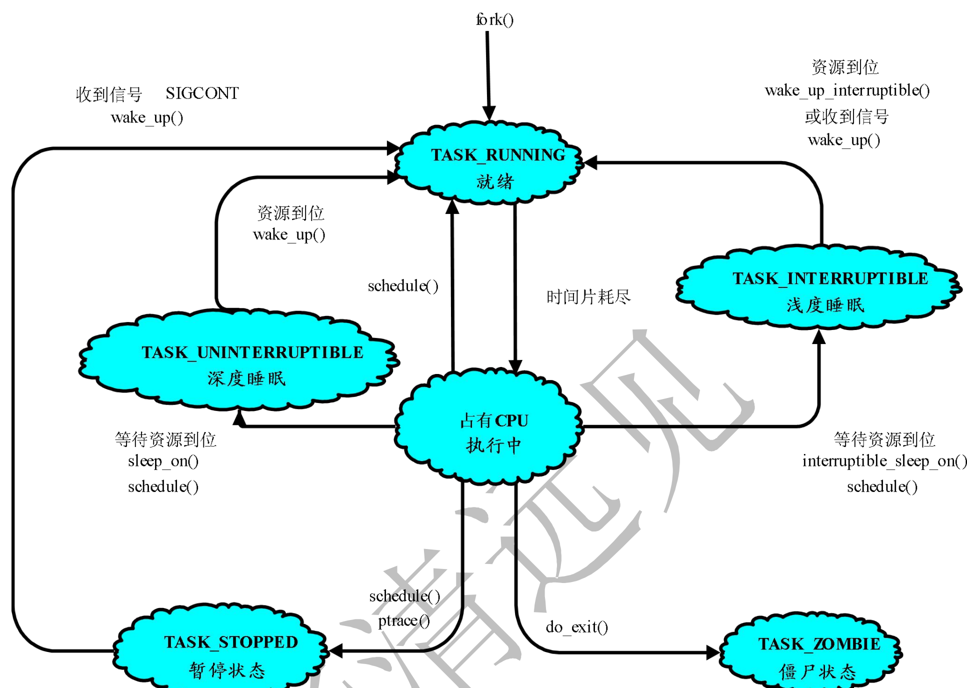


图 5.3 Linux 进程状态转换图

5.2.4 进程队列指针

为了有效地管理系统中的每个进程，需要采用一种高效的数据结构把系统内全部进程组织起来。当需要进程的信息时，可以从该结构中查找到相关进程。Linux 的所有进程组成一个双向链表，在内核中的类型是 struct list_head 的成员变量 tasks：

```
struct list_head tasks;
```

注意

链表是一种常用的组织有序数据的数据结构，它通过指针将一系列数据节点连接成一条数据链，是线性表的一种重要实现方式。相对于数组，链表具有更好的动态性，建立链表时无须预先知道数据总量，可以随机分配空间，可以高效地在链表中的任意位置实时插入或删除数据。链表的开销主要是访问的顺序性和组织链的空间损失。

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

list_head 结构体在 include/linux/list.h 文件中定义，代码如下：

```
struct list_head {
    struct list_head *next, *prev;
};
```

include/linux/list.h 文件中提供了用来操作该链表的函数和宏，它们实现了对链表的插入、删除、转移、合并、遍历。这些函数和宏比较简单，这里不再详细分析。

```
    static inline void list_add(struct list_head *new, struct list_head *head)
    static inline void list_add_tail(struct list_head *new, struct list_head *head)
    static inline void list_replace(struct list_head *old,struct list_head *new)
    static inline void list_replace_init(struct list_head *old,struct list_head *new)
    static inline void list_del_init(struct list_head *entry)
    static inline void list_move(struct list_head *list, struct list_head *head)
    static inline void list_move_tail(struct list_head *list,struct list_head *head)
    static inline int list_is_last(const struct list_head *list,const struct list_head
*head)
    static inline int list_empty(const struct list_head *head)
    static inline void list_splice(const struct list_head *list,struct list_head
*head)
    static inline void list_splice_tail(struct list_head *list,struct list_head *head)
    static inline void list_splice_init(struct list_head *list,struct list_head *head)
    static inline void list_splice_tail_init(struct list_head *list,struct list_head
*head)

    #define list_entry(ptr, type, member) \
        container_of(ptr, type, member)
    #define list_first_entry(ptr, type, member) \
        list_entry((ptr)->next, type, member)
    #define __list_for_each(pos, head) \
        for (pos = (head)->next; pos != (head); pos = pos->next)
    #define list_for_each_prev(pos, head) \
        for (pos = (head)->prev; prefetch(pos->prev), pos != (head); \
            pos = pos->prev)
    #define list_for_each_safe(pos, n, head) \
        for (pos = (head)->next, n = pos->next; pos != (head); \
            pos = n, n = pos->next)
    #define list_for_each_prev_safe(pos, n, head) \
        for (pos = (head)->prev, n = pos->prev; \
            prefetch(pos->prev), pos != (head); \
            pos = n, n = pos->prev)
    #define list_for_each_entry(pos, head, member)              \
        for (pos = list_entry((head)->next, typeof(*pos), member);    \
            prefetch(pos->member.next), &pos->member != (head);  \
            pos = list_entry(pos->member.next, typeof(*pos), member))
    #define list_for_each_entry_reverse(pos, head, member)          \
        for (pos = list_entry((head)->prev, typeof(*pos), member);    \
            prefetch(pos->member.prev), &pos->member != (head);  \
            pos = list_entry(pos->member.prev, typeof(*pos), member))
    #define list_prepare_entry(pos, head, member) \
        ((pos) ? : list_entry(head, typeof(*pos), member))
    #define list_for_each_entry_continue(pos, head, member)       \
```


华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com




```
    for (pos = list_entry(pos->member.next, typeof(*pos), member);    \
         prefetch(pos->member.next), &pos->member != (head);  \
         pos = list_entry(pos->member.next, typeof(*pos), member))
#define list_for_each_entry_continue_reverse(pos, head, member)        \
    for (pos = list_entry(pos->member.prev, typeof(*pos), member);    \
         prefetch(pos->member.prev), &pos->member != (head);  \
         pos = list_entry(pos->member.prev, typeof(*pos), member))
#define list_for_each_entry_from(pos, head, member)           \
    for (; prefetch(pos->member.next), &pos->member != (head);    \
         pos = list_entry(pos->member.next, typeof(*pos), member))
#define list_for_each_entry_safe(pos, n, head, member)              \
    for (pos = list_entry((head)->next, typeof(*pos), member),    \
         n = list_entry(pos->member.next, typeof(*pos), member);  \
         &pos->member != (head);                          \
         pos = n, n = list_entry(n->member.next, typeof(*n), member))
#define list_for_each_entry_safe_continue(pos, n, head, member)        \
    for (pos = list_entry(pos->member.next, typeof(*pos), member), \
         n = list_entry(pos->member.next, typeof(*pos), member);       \
         &pos->member != (head);                               \
         pos = n, n = list_entry(n->member.next, typeof(*n), member))
#define list_for_each_entry_safe_from(pos, n, head, member)            \
    for (n = list_entry(pos->member.next, typeof(*pos), member);       \
         &pos->member != (head);                               \
         pos = n, n = list_entry(n->member.next, typeof(*n), member))
#define list_for_each_entry_safe_reverse(pos, n, head, member)         \
    for (pos = list_entry((head)->prev, typeof(*pos), member),    \
         n = list_entry(pos->member.prev, typeof(*pos), member);  \
         &pos->member != (head);                          \
         pos = n, n = list_entry(n->member.prev, typeof(*n), member))
```

内核提供了宏 for_each_process，可以方便地搜索所有进程。当需要对系统中每一个进程进行操作时，该宏定义非常有用。它也在 sched.h 文件中定义，代码如下：

```
#define next_task(p)  list_entry(rcu_dereference((p)->tasks.next), struct task_struct, tasks)
#define for_each_process(p) \
for (p = &init_task ; (p = next_task(p)) != &init_task ; )
```

其中，init_task 是系统初始化过程中创建的第一个进程的进程描述符（PID），通过 list_entry()函数找到 task_struct 的地址，这样就可以打印进程的 PID 等域。

下面通过一个例子，进一步了解 list_head 结构的使用。下面的程序利用 list_head 结构遍历系统中的全部进程。

```
static int hello_init(void)
{
     struct task_struct *task,*p;
     struct list_head *pos;
     int count=0;
     printk("Hello World\n");
     task=&init_task;
     list_for_each(pos,&task->tasks)
             {
             p=list_entry(pos, struct task_struct, tasks);
```


华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

```
                count++;
                printk("%d-->%s\n",p->pid,p->comm);
                }
        printk("Total process is:%d\n",count);
        return 0;
}
static void hello_exit(void)
{
   printk( "Bye world!\n");
}
module_init(hello_init);
module_exit(hello_exit);
```

5.2.5 进程队列的全局变量

在 Linux 中，内核使用 current 变量表示当前正在运行的进程。current 是一个 task_struct 类型的全局变量。下面的语句可以打印当前进程。

```
printk( "Current task is %s [%d], current->comm, current->pid );
```

打开 arch/arm/include/asm/current.h 头文件，可以了解到 current 只是一个宏定义：

```
static inline struct task_struct *get_current(void) __attribute_const__;
static inline struct task_struct *get_current(void)
{
     return current_thread_info()->task;
}
#define current (get_current())
```

current_thread_info 函数用来获得当前进程的地址信息，代码及解释如下：

```
static inline struct thread_info *current_thread_info(void)
{
     register unsigned long sp asm ("sp");
     return (struct thread_info *)(sp & ~(THREAD_SIZE - 1));
}
```

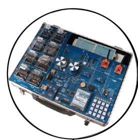
在 ARM 平台中，内核栈是 8KB，在栈顶（最低地址）处存放了一个指向当前进程的 task_struct 的指针，而 sp 就是当前的内核态堆栈指针，把它和~8191 进行“与”操作（清空最后 13 位），即获得栈顶的地址。然后把里面的值赋给 current，这样 current 就得到了当前 task_struct 的指针。THREAD_SIZE 在 thread_info.h 文件中有定义如下：

```
#define THREAD_SIZE        8192
#define THREAD_START_SP    (THREAD_SIZE - 8)
```

thread_info.h 文件中定义的 thread_info 结构如下：

```
struct thread_info {
     struct task_struct    *task;        /* main task structure */
     struct exec_domain    *exec_domain;/* execution domain */
     unsigned long     flags;       /* low level flags */
     __u32         cpu;         /* current CPU */
     int           preempt_count;    /* 0 => preemptable,<0 => BUG */
```


华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com




```
        mm_segment_t        addr_limit;   /* thread address space:
                                          0-0xBFFFFFFF for user
                                          0-0xFFFFFFFF for kernel */
        struct restart_block    restart_block;
        struct thread_info    *real_thread;    /* Points to non-IRQ stack */
};
```

其实 thread_info 结构的第一个成员就是一个指向 task_struct 结构的指针，所以要用 current_thread_info()->task 表示 task_struct 的地址，但是整个过程对用户是完全透明的，我们还是可以用 current 表示当前进程。

5.3 Linux 进程的创建



一个进程不会平白无故地诞生，它总会有自己的"父母"。在 Linux 中，通过调用 fork 系统调用来创建一个新的进程。新创建的子进程同样也能执行 fork，所以，有可能形成一颗完整的进程树。注意，每个进程只有一个父进程，但可以有 0 个、1 个、2 个或多个子进程。图 5.4 描述了 Linux 进程层次结构。其中，Pa 和 Pb 是进程 P 的两个子进程，而 Pc 和 Pd 又是进程 Pa 的两个子进程。

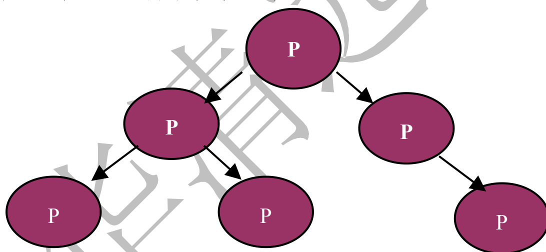


图 5.4　Linux 进程层次结构

Linux 在启动时就创建一个称为 init 的特殊进程，其进程标识符（PID）为 1，它是用户态下所有进程的祖先进程，以后诞生的所有进程都是它的子进程——或是它的儿子，或是它的孙子。1 号进程运行时查询系统当前存在的终端数，然后为每个终端创建一个新的管理进程，这些进程在终端上等待着用户的登录。当用户正确登录后，系统再为每一个用户启动一个 Shell 进程，由 Shell 进程等待并接受用户输入的命令。

应用程序可以通过 fork、vfork 或 clone 函数建立新的用户线程，这些函数分别通过系统调用访问 sys_fork、sys_vfork，或 sys_clone 内核函数建立新线程，而 sys_fork、sys_vfork 与 sys_clone 共同的调用函数为 do_fork。sys_fork 和 sys_vfork 在 arch/um/kernel/syscall.c 文件中实现，sys_clone 函数在 arch/um/sys-i386/syscall.c 文件中实现，代码如下：

```
long sys_fork(void)
{
```


华清远见 HQYJ.COM　华清远见教育集团官网：www.hqyj.com

华清远见

```
	long ret;

	current->thread.forking = 1;
	ret = do_fork(SIGCHLD, UPT_SP(&current->thread.regs.regs),
		      &current->thread.regs, 0, NULL, NULL);
	current->thread.forking = 0;
	return ret;
}

long sys_vfork(void)
{
	long ret;

	current->thread.forking = 1;
	ret = do_fork(CLONE_VFORK | CLONE_VM | SIGCHLD,
		      UPT_SP(&current->thread.regs.regs),
		      &current->thread.regs, 0, NULL, NULL);
	current->thread.forking = 0;
	return ret;
}

long sys_clone(unsigned long clone_flags, unsigned long newsp,
	       int __user *parent_tid, void *newtls, int __user *child_tid)
{
	long ret;

	if (!newsp)
		newsp = UPT_SP(&current->thread.regs.regs);

	current->thread.forking = 1;
	ret = do_fork(clone_flags, newsp, &current->thread.regs, 0, parent_tid,
		      child_tid);
	current->thread.forking = 0;
	return ret;
}
```

Linux 创建进程的函数的层次结构如图 5.5 所示。

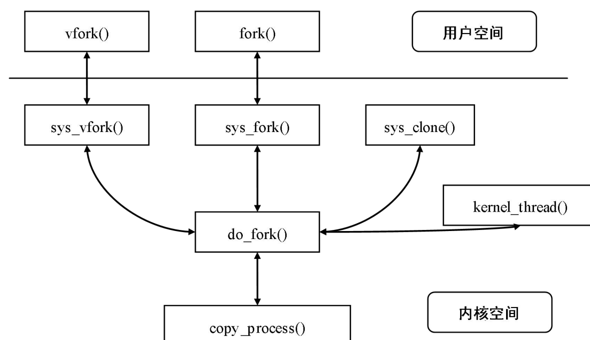
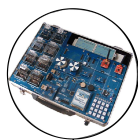


图 5.5　创建进程的函数的层次结构



从图 5.5 中可以看到，do_fork 函数是进程创建的基础，处理 clone、fork、vfork 系统调用。do_fork 使用辅助函数 copy_process 建立进程描述符及内核数据结构。可以在 kernel/fork.c 文件内找到 do_fork 函数原型（copy_process 函数也在该文件中）。

```
long do_fork(unsigned long clone_flags, //克隆标识
          unsigned long stack_start,         //栈起始位置
           //指向通用寄存器值的指针，通用寄存器的值是在用户态切换到内核态时保存到内核态堆栈
          struct pt_regs *regs,
          unsigned long stack_size,          //栈大小，一般设置为 0
          int __user *parent_tidptr,         //父进程用户态变量地址
          int __user *child_tidptr)          //子进程用户态变量地址
```

do_fork 函数中比较重要的有两个部分，首先是给子进程分配进程号，然后利用具体的进程号创建子进程。下面对这个函数进行简要分析。

```
{
    struct task_struct *p;          //进程描述符
    int trace = 0;
    long nr;
    if (unlikely(clone_flags & CLONE_STOPPED))
{
         static int __read_mostly count = 100;
         if (count > 0 && printk_ratelimit()) {
              char comm[TASK_COMM_LEN];
              count--;
              printk(KERN_INFO "fork(): process `%s' used deprecated "
                        "clone flags 0x%lx\n",
                   get_task_comm(comm, current),
                   clone_flags & CLONE_STOPPED);
         }
    }
```

在 2.6 内核中随处可以看到 likely()和 unlikely()宏，这两个宏在内核中定义如下：

```
#define likely(x)      __builtin_expect(!!(x), 1)
#define unlikely(x)        __builtin_expect(!!(x), 0)
```

__builtin_expect() 是 GCC (version>=2.96）提供给程序员使用的，目的是将“分支转移”的信息提供给编译器，这样，编译器就可以对代码进行优化，以减少指令跳转带来的性能下降。

__builtin_expect((x),1)表示 x 的值为真的可能性更大。

__builtin_expect((x),0)表示 x 的值为假的可能性更大。

也就是说，使用 likely()，执行 if 后面的语句的机会更大；使用 unlikely()，执行 else 后面的语句的机会更大。

回到 do_fork 函数中，CLONE_STOPPED 位表示设置进程为停止状态（类似的 clone 标志位在 sched.h 中定义），因此，为假的可能性更大。随后内核会通过 get_task_comm 函数获得当前进程（current）的命令名，显示给用户。

```
         if (likely(user_mode(regs)))
         trace = tracehook_prepare_clone(clone_flags);
```


华清远见 HQYJ.COM

华清远见教育集团官网：www.hqyj.com

上面的代码检查子进程是否为内核线程，如果不是，则将 clone_flags 设为 0。

```
p = copy_process(clone_flags, stack_start, regs, stack_size,
     child_tidptr, NULL, trace);
```

随后内核调用 copy_process 函数复制进程描述符。如果成功，该函数将返回刚创建的 task_struct 描述符的地址。

```
     if (!IS_ERR(p)) {
          struct completion vfork;   //completion 同步机制
          trace_sched_process_fork(current, p);
          nr = task_pid_vnr(p);      //获得 PID
          if (clone_flags & CLONE_PARENT_SETTID)
               put_user(nr, parent_tidptr);
          if (clone_flags & CLONE_VFORK) {
               p->vfork_done = &vfork;
               init_completion(&vfork);
          }
          audit_finish_fork(p);
          tracehook_report_clone(trace, regs, clone_flags, nr, p);
```

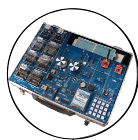
上面的语句判断是否设置 CLONE_VFORK 标志，如果设置，则初始化进程描述符的 vfork_done 标志，然后初始化 completion。

```
          p->flags &= ~PF_STARTING;
          if (unlikely(clone_flags & CLONE_STOPPED)) {
               /*
                * We'll start up with an immediate SIGSTOP.
                */
               sigaddset(&p->pending.signal, SIGSTOP);
               set_tsk_thread_flag(p, TIF_SIGPENDING);
               __set_task_state(p, TASK_STOPPED);
          } else {
               wake_up_new_task(p, clone_flags);
          }
          tracehook_report_clone_complete(trace, regs,clone_flags, nr, p);
```

如果子进程被跟踪或子进程初始化成 STOP 状态，则发送 SIGSTOP 信号。由于子进程现在还没有运行，信号不能被处理，所以设置 TIF_SIGPENDING 标志；如果设置了 CLONE_STOPPED 标志或必须跟踪子进程，则将子进程的状态设置为 TASK_STOPPED，并为子进程增加挂起的 SIGSTOP 信号；如果没有设置 CLONE_STOPPED 标志，则调用 wake_up_new_task 函数来处理父进程和子进程。

```
          if (clone_flags & CLONE_VFORK) {
               freezer_do_not_count();
               wait_for_completion(&vfork);
               freezer_count();
               tracehook_report_vfork_done(p, nr);
          }
          else {
          nr = PTR_ERR(p);
          }
          return nr;
```


华清远见 HQYJ.COM 华清远见教育集团官网：www.hqyj.com



如果设置了 CLONE_VFORK 标志，则挂起父进程直到子进程释放自己的内存地址空间，也就是说，直到子进程结束或执行新的程序。wait_for_completion(&vfork)用来等待子进程释放自己的内存地址空间。

5.4 Linux 进程相关的系统调用



由于 Linux 以分裂的方法来创建新进程，这样就使得系统中的所有进程都有亲属关系，这种亲属关系也会给进程的运行带来一定的影响。为了表示一个进程的亲属关系，每个进程控制块中都有记录这些亲属关系的成员，代码如下所示：

```
struct task_struct {
…
struct task_struct *real_parent;    /* 父进程 */
     struct task_struct *parent;    /* 养父进程*/
     /*
      * children/sibling forms the list of my natural children
      */
     struct list_head children;     /* 子进程列表 */
     struct list_head sibling;      /* linkage in my parent's children list */
…
};
```

从内核代码中可以看到，进程有父进程和养父进程。那么养父进程是什么呢？

当一个进程被创建出来以后，其控制块中的指针 real_parent 和 parent 都指向同一个进程→父进程。但是当其他某个进程通过系统调用 ptrace()跟踪这个进程时，被跟踪进程的 parent 会指向这个跟踪进程，而这个跟踪进程也有指针指向被跟踪进程，以便通过这个指针得到被跟踪进程的控制块以获得被跟踪进程的信息，所以这个跟踪进程有些类似“监管人”，因此这个进程也称为“养父”进程。

5.4.1 execve()系统调用

Linux 提供了 execl、execlp、execle、execv、execvp 和 execve 共 6 个用以执行一个可执行文件的函数（统称为 exec 函数）。这些函数的不同之处在于对命令行参数和环境变量参数的传递方式不同。每个函数的第一个参数都是要被执行的程序的路径，第二个参数则向程序传递了命令行参数，第三个参数则向程序传递环境变量。以上函数的本质都是调用 arch/x86/kernel/process.c 文件中实现的系统调用 sys_execve 来执行一个可执行文件，该函数代码如下：

```
long sys_execve(char __user *name, char __user * __user *argv,char __user * __user
*envp, struct pt_regs *regs)
{
     long error;
```


华清远见 HQYJ.COM 华清远见教育集团官网：www.hqyj.com

```
        char *filename;

        filename = getname(name);
        error = PTR_ERR(filename);
        if (IS_ERR(filename))
            return error;
        error = do_execve(filename, argv, envp, regs);

    #ifdef CONFIG_X86_32
        if (error == 0) {
            /* Make sure we don't return using sysenter.. */
                set_thread_flag(TIF_IRET);
          }
    #endif

        putname(filename);
        return error;
    }
```

从函数原型提供的参数可以看出，这个函数接收的参数都来自用户空间。在 sys_execve 中调用了 execve 函数，该函数也在 arch/um/kernel/exec.c 中实现。从下面的代码中可以了解到，execve 实质上调用了 do_execve 函数。

```
    int do_execve(char * filename,
        char __user *__user *argv,
        char __user *__user *envp,
        struct pt_regs * regs)
    {
        struct linux_binprm *bprm;
        struct file *file;
        struct files_struct *displaced;
        bool clear_in_exec;
        int retval;

        retval = unshare_files(&displaced);
        if (retval)
            goto out_ret;

        retval = -ENOMEM;
        bprm = kzalloc(sizeof(*bprm), GFP_KERNEL);
        if (!bprm)
            goto out_files;

        retval = prepare_bprm_creds(bprm);
        if (retval)
            goto out_free;

        retval = check_unsafe_exec(bprm);
        if (retval < 0)
            goto out_free;
        clear_in_exec = retval;
        current->in_execve = 1;
```

```
	file = open_exec(filename);
	retval = PTR_ERR(file);
	if (IS_ERR(file))
		goto out_unmark;

	sched_exec();

	bprm->file = file;
	bprm->filename = filename;
	bprm->interp = filename;

	retval = bprm_mm_init(bprm);
	if (retval)
		goto out_file;

	bprm->argc = count(argv, MAX_ARG_STRINGS);
	if ((retval = bprm->argc) < 0)
		goto out;

	bprm->envc = count(envp, MAX_ARG_STRINGS);
	if ((retval = bprm->envc) < 0)
		goto out;

	retval = prepare_binprm(bprm);
	if (retval < 0)
		goto out;

	retval = copy_strings_kernel(1, &bprm->filename, bprm);
	if (retval < 0)
		goto out;

	bprm->exec = bprm->p;
	retval = copy_strings(bprm->envc, envp, bprm);
	if (retval < 0)
		goto out;

	retval = copy_strings(bprm->argc, argv, bprm);
	if (retval < 0)
		goto out;

	current->flags &= ~PF_KTHREAD;
	retval = search_binary_handler(bprm,regs);
	if (retval < 0)
		goto out;

	/* execve succeeded */
	current->fs->in_exec = 0;
	current->in_execve = 0;
	acct_update_integrals(current);
	free_bprm(bprm);
	if (displaced)
		put_files_struct(displaced);
	return retval;
```

```
out:
    if (bprm->mm)
        mmput (bprm->mm);

out_file:
    if (bprm->file) {
        allow_write_access(bprm->file);
        fput(bprm->file);
    }

out_unmark:
    if (clear_in_exec)
        current->fs->in_exec = 0;
    current->in_execve = 0;

out_free:
    free_bprm(bprm);

out_files:
    if (displaced)
        reset_files_struct(displaced);
out_ret:
    return retval;
}
```

上述代码中有两个函数：copy_strings_kernel 和 copy_strings。copy_strings 把参数及执行的环境从用户空间复制到内核空间的 bprm 变量中，而调用 copy_strings_kernel()从内核空间中复制文件名。

这里还提到了 linux_binfmt 数据结构（/include/linux/binfmt.h)，它用来支持各种文件系统，所以 Linux 中的 exec()函数在执行时，使用已注册的 linux_binfmt 结构就可以支持不同的二进制格式，即多种文件系统（Ext3、FAT 等）。需要指出的是，binux_binfmt 结构中嵌入了两个指向函数的指针，一个指针指向可执行代码，另一个指向库函数；使用这两个指针是为了装入可执行代码和要使用的库。linux_binfmt 结构描述如下：

```
struct linux_binprm{
    char buf[BINPRM_BUF_SIZE];
#ifdef CONFIG_MMU
    struct vm_area_struct *vma;
#else
# define MAX_ARG_PAGES     32
    struct page *page[MAX_ARG_PAGES];
#endif
    struct mm_struct *mm;
    unsigned long p; /* current top of mem */
    unsigned int sh_bang:1,
            misc_bang:1;
#ifdef __alpha__
    unsigned int taso:1;
#endif
    unsigned int recursion_depth;
```


华清远见
HQYJ.COM
华清远见教育集团官网：www.hqyj.com

```
        struct file * file;
        int e_uid, e_gid;
        kernel_cap_t cap_post_exec_permitted;
        bool cap_effective;
        void *security;
        int argc, envc;
        char * filename;      /* Name of binary as seen by procps */
        char * interp;        /* Name of the binary really executed. Most
                       of the time same as filename, but could be
                       different for binfmt_{misc,script} */
        unsigned interp_flags;
        unsigned interp_data;
        unsigned long loader, exec;
    };
```

在 do_execve 函数中有一个参数值得我们注意——*regs。它是 pt_regs 类型的数据结构，该参数描述了在执行该系统调用时，用户态下的 CPU 寄存器在核心态的栈中的保存情况。通过这个参数，sys_execve 能获得保存在用户空间的以下信息：可执行文件路径的指针（regs.ebx 中）、命令行参数的指针（regs.ecx 中）和环境变量的指针（regs.edx 中）。代码如下：

```
    struct pt_regs {
        long ebx;
        long ecx;
        long edx;
        long esi;
        long edi;
        long ebp;
        long eax;
        int  xds;
        int  xes;
        int  xfs;
        /* int  gs; */
        long orig_eax;
        long eip;
        int  xcs;
        long eflags;
        long esp;
        int  xss;
    };
```

回顾 do_execve()，它的工作流程如下：

（1）按参数 filename 找到相关文件的节点，保存与文件相关的数据，并检查它的存取权限。

（2）检查文件格式，并找到能打开这种格式的装载函数。

（3）确认文件的可执行权限后，首先放弃子进程继承自父进程的虚拟空间结构，调用 do_mmap()函数，根据文件中的虚拟地址信息建立自己的虚拟空间描述结构，建立空页表，并使其映射到子进程的进程虚拟地址空间。

（4）创建运行环境。

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

（5）在子进程运行过程中，由 Linux 的请页机制逐步为其分配所需要的物理内存空间。

下面看一个运行子进程的程序。

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main()
{
        char * argu_list[]={"ls","/","-l",NULL};
        int status=0;
        pid_t child_pid=fork();
        if(child_pid==0)
        {
                //子进程
                execvp("ls",argu_list);
                fprintf(stderr,"exec ls error\n");
                exit(1);
        }
        else
        {
                wait(&status);
                if(WIFEXITED(status))
                        printf("the child process exit normally.\n");
                else
                        printf("the child process exit abnormally.\n");
                exit(0);
        }
}
```

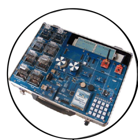
5.4.2 wait()系统调用

为了方便用户处理父进程与子进程之间的一些事物，Linux 允许父进程在创建了进程之后，通过调用 wait()先进入等待状态，以使子进程先运行，然后再决定自己的进一步行为，这种方式称为父进程的阻塞方式。那么，wait()在什么时候用呢？在一些情况下，父进程可能比子进程结束得要早。如果父进程提前结束了，子进程就变成了僵尸进程。我们需要父进程来清理子进程结束后的一些环境。这时调用 wait，父进程将阻塞在 wait()处，等待子进程结束。

5.4.3 exit()系统调用

进程的结束可以用 exit()或_exit()系统调用。无论在程序中的什么位置，只要执行到 exit 系统调用，进程就会停止剩下的所有操作，清除包括 PCB 在内的各种数据结构，并终止本进程的运行。对系统调用而言，_exit 和 exit 是一对孪生兄弟，它们最大的区别就在于 exit()函数在调用之前要检查文件的打开情况，把文件缓冲区中的内容写回文件；

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com



而_exit()直接使进程停止运行，清除其使用的内存空间，并销毁其在内核中的各种数据结构，如图 5.6 所示。

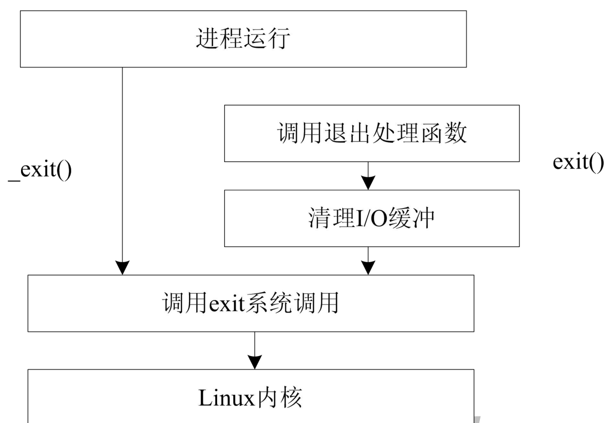


图 5.6　exit 和_exit 的区别

下面通过一个简单的程序了解一下 exit 系统调用的用法。

```
main()
{
printf("Process will be exit!\n");
exit(0);
printf("You cannot see this line!\n");
}
```

编译运行该程序可以看到，程序并没有打印后面的“You cannot see this line!”，因为在此之前，在执行到 exit(0)时，进程就已经终止了。

5.5 Linux 的进程调度



操作系统的职能之一是管理并调度系统中的进程，而衡量一个操作系统的关键之一就是调度算法。一个优秀的调度算法能够充分而高效地利用系统资源。存在于 Linux 中的进程通过 Linux 调度程序被调度。每个进程的调度策略在 task_struct 中规定（policy 属性），Linux 内核提供的调度策略类型如表 5.4 所示。其中，SCHED_FIFO 和 SCHED_RR 的优先级高于所有 SCHED_NORMAL 的进程。

提示

决定什么时候以何种方式选择一个进程运行的这组规则就是调度策略。

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

表 5.4　Linux 进程调度策略

调度策略	采用的调度算法
SCHED_NORMAL	非实时，基于优先级的轮转法
SCHED_FIFO	实时，先进先出算法
SCHED_RR	实时，基于优先级的轮转法
SCHED_BATCH	与 SCHED_NORMAL 类似，该调度算法将使得调度器假定进程对 CPU 时间要求较多
SCHED_ISO	目前还没有实现
SCHED_IDLE	进程优先级降为最低

为了实现进程的调度，每一个多任务操作系统都会为不同的任务分配一个任务优先级，进程之间是竞争资源（如 CPU 和内存的占用）关系，这个竞争优劣是通过一个数值来实现的，也就是谦让度。高谦让度表示进程优先级别最低，负值或 0 表示高优先级，对其他进程不谦让，也就是拥有优先占用系统资源的权力。谦让度的值从-20 到 19。

目前，硬件技术发展迅速，大多情况下不必设置进程的优先级，除非在进程失控而疯狂占用资源的情况下，才设置优先级。在迫不得已的情况下，可以杀掉失控进程。Linux 系统提供了 nice 命令来完成进程优先级调整。例如，下面的命令将运行 vim 程序，并为它指定谦让度增量为 6：

```
[root@localhost ~]# nice -n 6 vim
```

nice 的最常用的应用包括：

```
nice -n 谦让度的增量值 程序
renice 是通过进程 ID（PID）来改变谦让度，进而达到更改进程的优先级。
renice 谦让度 PID
renice 所设置的谦让度就是进程的绝对值。
```

2.6 版本的 Linux 内核提供了 140 个优先级，后 40 个和 nice 值一一对应，属于 SCHED_NORMAL 调度策略，前 100 个属于 SCHED_FIFO 和 SCHED_RR 策略。

在调度算法的实现上，Linux 中的每个任务有 4 个与调度相关的参数，它们是 rt_priority、policy、priority（nice）、counter。调度程序根据这 4 个参数进行进程调度。

在 SCHED_NORMAL 调度策略中，调度器总是选择那个 priority+counter 值最大的进程来调度执行。从逻辑上进行分析，SCHED_NORMAL 调度策略存在着调度周期（epoch），在每一个调度周期中，一个进程的 priority 和 counter 值的大小影响了当前时刻应该调度哪一个进程来执行，其中 priority 是一个固定不变的值，在进程创建时就已经确定了，它代表该进程的优先级，也代表该进程在每一个调度周期中能够得到的时间片的多少；counter 是一个动态变化的值，它反映了一个进程在当前的调度周期中还剩下的时间片。在每一个调度周期的开始，priority 的值被赋给 counter，然后每次该进程被调度执行时，counter 值都减少。当 counter 值为零时，该进程用完自己在本调度周期中的时间片，不再参与本调度周期的进程调度。当所有进程的时间片都用完时，一个调度周期结束，这样周而复始。另外，可以看出 Linux 系统中的调度周期不是静态的，它是

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com



一个动态变化的量，例如，处于可运行状态的进程的多少和它们 priority 的值都可以影响一个 epoch 的长短。在 2.4 以上的内核中，priority 被 nice 所取代，但二者作用类似。

在 SCHED_FIFO 调度策略中，不同的进程根据静态优先级进行排队；然后在同一优先级的队列中，谁先准备好运行就先调度谁，并且正在运行的进程不会被终止，直到以下情况发生：

- 被更高优先级的进程强占 CPU。
- 自己因为资源请求而阻塞。
- 自己主动放弃 CPU（调用 sched_yield）。

SCHED_RR 与上面的 SCHED_FIFO 类似，不同之处在于它给每个进程分配一个时间片，时间片到了正在执行的进程就放弃执行；时间片的长度可以通过 sched_rr_get_interval 调用得到。

5.6 实时 Linux



Linux 是一种非实时操作系统，在某些对实时性要求比较严格的嵌入式应用中，Linux 系统的实时性一直是最大的顽症。不过，也有人陆续推出 Linux 内核的实时补丁，在一定程度上解决了实时性的问题。其中，最有名的是 RT-Linux。

RT-Linux 是新墨西哥科技大学的研究成果，它的基本思想是，为了在 Linux 系统中提供对于硬实时的支持，实现了一个微内核的小的实时操作系统，而将普通 Linux 系统作为一个该操作系统中的一个低优先级的任务来运行。另外，普通 Linux 系统中的任务可以通过 FIFO 和实时任务进行通信。RT-Linux 的框架如图 5.7 所示。

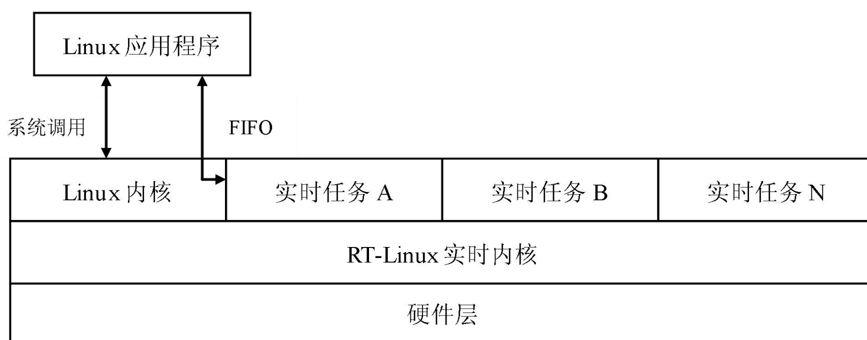


图 5.7　RT-Linux 结构

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

RT-Linux 的关键技术是通过软件来模拟硬件的中断控制器。当 Linux 系统要封锁 CPU 的中断时，RT-Linux 中的实时子系统会截取到这个请求，把它记录下来，而实际上并不真正封锁硬件中断，这样就避免了由于封锁中断所造成的系统在一段时间没有响应的情况，从而提高了实时性。当有硬件中断到来时，RT-Linux 截取该中断，并判断是否有实时子系统中的中断例程来处理，还是传递给普通的 Linux 内核进行处理。另外，普通 Linux 系统中的最小定时精度由系统中的实时时钟的频率决定，一般 Linux 系统将该时钟设置为每秒 100 个时钟中断，所以 Linux 系统中一般的定时精度为 10ms，即时钟周期是 10ms，而 RT-Linux 通过将系统的实时时钟设置为单次触发状态，可以提供十几个微秒级的调度粒度。

RED-Linux 是另外一种实时 Linux 系统，由加州大学 Irvine 分校开发。它将对实时调度的支持和 Linux 很好地实现在同一个操作系统内核中。它同时支持 3 种类型的调度算法，即：Time-Driven、Priority-Driven、Share-Driven。RED-Linux 的设计目标就是提供一个可以支持各种调度算法的通用调度框架，该系统给每个任务增加了如下几项属性，并将它们作为进程调度的依据。

- Priority：作业的优先级。
- Start-Time：作业的开始时间。
- Finish-Time：作业的结束时间。
- Budget：作业在运行期间所要使用的资源数量。

通过调整这些属性的取值及调度程序按照什么样的优先顺序来使用这些属性值，几乎可以实现所有的调度算法。这样，可以将 3 种不同的调度算法无缝、统一地结合到一起。RED-Linux 调度程序的框架结构如图 5.8 所示。

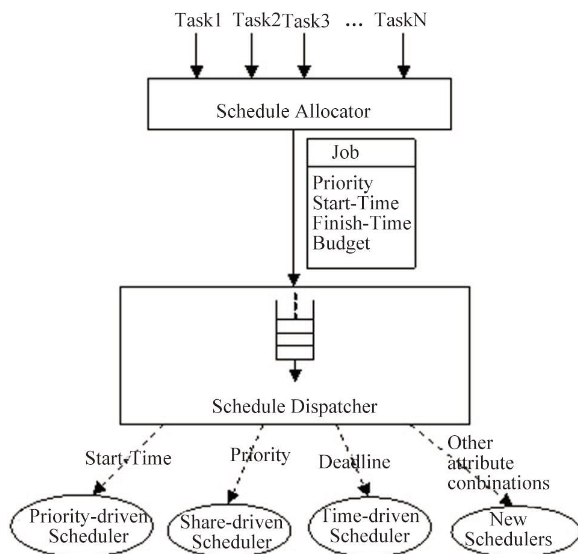
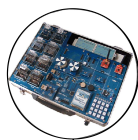


图 5.8　RED-Linux 调度框架

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com



5.7 本章习题



1．和进程管理相关的内核文件都有哪些？
2．什么是进程和线程？
3．什么是进程描述符？怎样得到当前进程的进程描述符？
4．进程的内核栈有多大？
5．进程的状态都有哪些？在什么情况下发生转化？
6．Linux中所有进程之间的关系是怎么样的？

华清远见 HQYJ.COM

华清远见

第6章 进程间通信

进程间通信就是在不同进程之间传播或交换信息。在一个操作系统中，调节各个进程对资源的共享是操作系统的重要职能之一。从原理上看，进程通信的关键技术就是在进程间建立某种共享区，利用进程都可以访问共享区的特点来建立一些通信通道。本章介绍Linux操作系统提供的部分进程间通信方式。包括信号量、共享内存、消息队列及管道。

华清远见 HQYJ.COM

华清远见教育集团官网：www.hqyj.com



华清远见

华清远见
HQYJ.COM
华清远见教育集团官网：www.hqyj.com

6.1 什么是进程间通信



在单任务系统中，任务被线性执行时，不可能被抢占，所以不需要同步机制保护共享资源和临界资源，而且单任务也不存在数据交换的问题。但对于多任务操作系统来说，上述问题都是存在的。如何保护临界资源和进行数据交换，都是操作系统需要解决的问题。进程间通信（IPC）就是为了解决这些问题而提出的特有机制，它们为多任务系统提供了不同进程的通信机制，同时也提供了对于临界资源和共享资源的保护。

进程间通信的主要目的是实现同一计算机系统内部的相互协作的进程之间的数据共享与信息交换，由于这些进程处于同一软件和硬件环境下，利用操作系统提供的编程接口，用户可以方便地在程序中实现这种通信；应用程序间通信的主要目的是实现不同计算机系统中的相互协作的应用程序之间的数据共享与信息交换，由于应用程序分别运行在不同计算机系统中，它们之间要通过网络之间的协议才能实现数据共享与信息交换。

Linux 系统的进程通信方式基本上是从 UNIX 平台上的进程通信手段继承而来的。而对 UNIX 发展做出重大贡献的贝尔实验室及 BSD 在进程间通信方面的侧重点有所不同。前者对 UNIX 早期的进程间通信手段进行了系统的改进和扩充，形成了“System V IPC”，通信进程局限在单个计算机内；后者则跳过了该限制，形成了基于套接口（Socket）的进程间通信机制。Linux 则把两者都继承了下来，如图 6.1 所示。

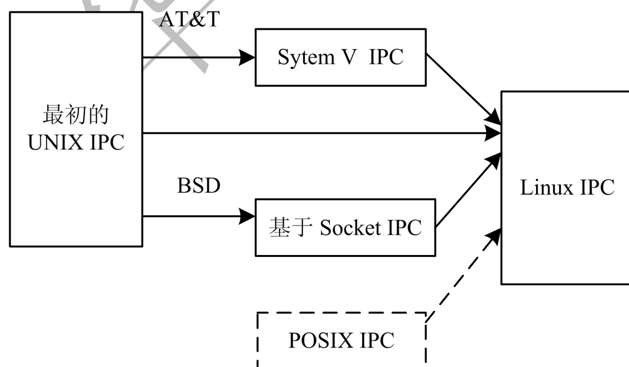


图 6.1 Linux 系统的通信方式

其中，最初 UNIX IPC 包括管道、FIFO、信号；System V IPC 包括 System V 消息队列、System V 信号灯、System V 共享内存区；Posix IPC 包括 Posix 消息队列、Posix 信号灯、Posix 共享内存区。

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

对于不同的嵌入式系统，进程间通信的实现方式有所不同，但是基本原理都差不多。对于进程间通信，主要有两种方式：虚拟内存系统中的进程间通信和 Falt 内存系统中的进程间通信。

μC/OS 是比较典型的 Falt 内存系统，它不支持虚拟内存机制，也没有用户空间和内核空间的区别，实际上它类似于 Linux 的内核空间，不同任务间可以相互访问，没有不同进程间内存保护机制。所以可以完全利用 Linux 系统中的同一进程中不同线程的通信机制。由于所有的任务与中断都共享同一地址空间，所以同步机制也与任务间通信在同一空间中实现，使这两种机制的相互替换成为可能。

Windows 作为一种复杂的多任务系统，也提供了多种进程间通信方式，包括文件映射、共享内存、匿名管道、命名管道、邮件槽、剪贴板、动态数据交换（DDE）、对象链接与嵌入（OLE）、动态链接库（DLL）、远程过程调用（RPC）、NetBios 函数、Sockets、WM_COPYDATA 消息。

本章将阐述 Linux 系统是如何实现进程间通信的。

6.2 互斥与同步



互斥与同步是进程间通信中非常重要的一对概念，也是相交进程之间的两种主要关系。在嵌入式操作系统开发中经常会遇到同步、互斥的问题，如果处理得不好，程序就会出现很多意想不到的结果。而在多处理器之间、ISR 与 ISR 之间、ISR 与任务之间、任务与任务之间都可能需要互斥与同步。例如，不同任务优先级的抢占、中断处理等。

互斥和同步是两个紧密相关而又容易混淆的概念。所谓互斥，是指某一资源同时只允许一个访问者对其进行访问，具有唯一性和排他性。但互斥无法限制访问者对资源的访问顺序，即访问是无序的。两个互斥的进程，只能等到前一个进程运行完后，下一个进程才能运行。所谓同步，是指在互斥的基础上（大多数情况），通过其他机制实现访问者对资源的有序访问。在大多数情况下，同步已经实现了互斥，特别是所有写入资源的情况必定是互斥的。少数情况是指可以允许多个访问者同时访问资源。

同步是一种更为复杂的互斥，而互斥是一种特殊的同步。

例如，下面的代码：

```
s=1;
 i=0;
codeostart
          p1();
          p2();
codeend
      p1(){
          while(){   //循环
              p(s);
```


华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

```
            i++;
            printf("%d",i);
            v(s);
          }
        }
    p2(){
        while(){   //循环
            p(s);
            i--;
            printf("%d",i);
            v(s);
          }
    }
```

其中，p1()和 p2()是两个进程。i++和 i--分别是两个程序片断。如果没有执行完 p1()是不允许执行 p2()的。这里的 p1()和 p2()并不是指一个完整的进程，只是两个进程的临界资源（共有）。

这里必须掌握“原子操作”的概念。所谓原子操作，就是该操作绝不会在执行完毕前被任何其他任务或事件打断，也就是说，它是最小的执行单位，不可能有比它更小的执行单位。这里的原子实际上是使用了物理学中的物质微粒的概念。原子操作主要用于实现资源计数，很多引用计数（Refcnt）就是通过原子操作实现的。

原子操作需要硬件的支持，因此是架构相关的，其 API 和原子类型的定义都定义在内核源代码的 include/asm-generic/atomic.h 文件中，它们都使用汇编语言实现，因为 C 语言并不能实现这样的操作。以 ARM 平台为例，原子类型定义如下：

```
typedef struct { volatile int counter; } atomic_t;
```

volatile 修饰字段告诉 GCC 编译器不要对该类型的数据进行优化处理，对它的访问都是对内存的访问，而不是对寄存器的访问。

有关原子操作的更多 API，可以阅读 atomic.h 文件，这里不再赘述。

```
#define atomic_inc_not_zero(v) atomic_add_unless((v), 1, 0)

#define atomic_add(i, v)   (void) atomic_add_return(i, v)
#define atomic_inc(v)      (void) atomic_add_return(1, v)
#define atomic_sub(i, v)   (void) atomic_sub_return(i, v)
#define atomic_dec(v)      (void) atomic_sub_return(1, v)

#define atomic_inc_and_test(v) (atomic_add_return(1, v) == 0)
#define atomic_dec_and_test(v) (atomic_sub_return(1, v) == 0)
#define atomic_inc_return(v)    (atomic_add_return(1, v))
#define atomic_dec_return(v)    (atomic_sub_return(1, v))
#define atomic_sub_and_test(i, v) (atomic_sub_return(i, v) == 0)
#define atomic_add_negative(i,v) (atomic_add_return(i, v) < 0)
```


华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

6.3 信号量



6.3.1 什么是信号量

信号量是最早出现的用来解决进程同步与互斥问题的机制，包括一个称为信号量的变量及对它进行的两个原语操作。这两个原语操作使用荷兰语命令：Prolagen（降低）和 Verhogen（升起），通常简称为 P、V 操作。

信号量可以用来保护两个或多个关键代码段，这些关键代码段不能并发调用。在进入一个关键代码段之前，线程必须获取一个信号量。如果关键代码段中没有任何线程，那么线程会立即进入该框图中的那个部分。一旦该关键代码段完成了，那么该线程必须释放信号量。其他想进入该关键代码段的线程必须等待直到第一个线程释放信号量。为了完成这个过程，需要创建一个信号量，然后将 Acquire Semaphore VI 及 Release Semaphore VI 分别放置在每个关键代码段的首末端。确认这些信号量 VI 引用的是初始创建的信号量。

下面看一下如何用信号量解决经典的生产者—消费者问题。问题描述如下：

一个仓库可以存放 *k* 件物品。生产者每生产一件产品，将产品放入仓库，仓库满了就停止生产。消费者每次从仓库中取一件物品，然后进行消费，仓库空时就停止消费。代码中，Producer 进程是生产者进程，Consumer 是消费者进程。共有的数据结构如下：

```
buffer: array [0..k-1] of integer;
in,out: 0..k-1;
```

其中，in 记录第一个空缓冲区，out 记录第一个不空的缓冲区。

```
s1,s2,mutex: semaphore;
```

其中，s1 控制缓冲区不满，s2 控制缓冲区不空，mutex 保护临界区。

初始化 s1=k，s2=0，mutex=1。

```
producer（生产者进程）:
 Item_Type item;
{
   while (true)
   {
     produce(&item);
     p(s1);
     p(mutex);
     buffer[in]:=item;
     in:=(in+1) mod k;
     v(mutex);
     v(s2);
   }
}
```


华清远见 HQYJ.COM　华清远见教育集团官网：www.hqyj.com

```
consumer（消费者进程）：
 Item_Type item;
{
   while (true)
   {
     p(s2);
     p(mutex);
      item:=buffer[out];
      out:=(out+1) mod k;
      v(mutex);
      v(s1);
     consume(&item);
   }
 }
```

6.3.2 信号量的内核实现

Linux 内核的信号量在概念和原理上与用户态的 System V 的 IPC 机制信号量是一样的，但是它只能在内核空间使用。信号量在创建时需要设置一个初始值，表示同时可以有几个任务访问该信号量保护的共享资源，初始值为 1 就变成互斥锁（Mutex），即同时只能有一个任务可以访问信号量保护的共享资源。当任务访问完被信号量保护的共享资源后，必须释放信号量，释放信号量通过把信号量的值加 1 实现，如果信号量的值为非正数，表明有任务等待当前信号量，所以，它也唤醒所有等待该信号量的任务。

在 Linux 内核源代码的 kernel/printk.c 中，使用宏 DECLARE_MUTEX 声明了一个互斥锁 console_sem，它用于保护 console 驱动列表 console_drivers 及同步对整个 console 驱动系统的访问。

信号量数据结构定义在 include/linux/semaphore.h 中，代码如下：

```
struct semaphore {
    spinlock_t        lock;
    unsigned int count;
    struct list_head  wait_list;
};
```

其中，wait _list 字段存放当前等待该信号量的所有进程的链表。如果 count 大于或等于 0，该链表就为空。

与信号量相关的内核实现还包括 sema_init 函数，其中的 val 表示信号量的初始值，代码如下：

```
static inline void sema_init(struct semaphore *sem, int val)
{
    static struct lock_class_key __key;
    *sem = (struct semaphore) __SEMAPHORE_INITIALIZER(*sem, val);
    lockdep_init_map(&sem->lock.dep_map, "semaphore->lock", &__key, 0);
}
#define init_MUTEX(sem)         sema_init(sem, 1)
#define init_MUTEX_LOCKED(sem) sema_init(sem, 0)
```


华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

在 Linux 系统中，P 函数被称为 down，指的是该函数减小了信号量的值。它也许会将调用者置于休眠状态，然后等待信号量变得可用，之后再授予调用者对被保护资源的访问权限。

down 函数有如下几个版本，这些函数都在 semaphore.c 文件中实现。

1. down

```
void down(struct semaphore *sem)
{
    unsigned long flags;

    spin_lock_irqsave(&sem->lock, flags);
    if (likely(sem->count > 0))
        sem->count--;
    else
        __down(sem);
    spin_unlock_irqrestore(&sem->lock, flags);
}
```

这个版本用来减小信号量的值，并在必要时一直等待。

2. down_interruptible

```
int down_interruptible(struct semaphore *sem)
{
    unsigned long flags;
    int result = 0;

    spin_lock_irqsave(&sem->lock, flags);
    if (likely(sem->count > 0))
        sem->count--;
    else
        result = __down_interruptible(sem);
    spin_unlock_irqrestore(&sem->lock, flags);

    return result;
}
```

作为通常规则，我们不应该使用非中断版本。使用该函数时，如果操作被中断，则该函数会返回非 0 值，而调用者不会拥有该信号量。因此，对该函数的正确使用需要始终检查返回值，并做出相应的响应。

3. down_killable

```
int down_killable(struct semaphore *sem)
{
    unsigned long flags;
    int result = 0;

    spin_lock_irqsave(&sem->lock, flags);
    if (likely(sem->count > 0))
        sem->count--;
    else
```


华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

```
        result = __down_killable(sem);
    spin_unlock_irqrestore(&sem->lock, flags);

    return result;
}
```

TASK_KILLABLE 是 Linux Kernel 2.6.25 引入了一种新的进程睡眠状态。相对应的，down 函数有一个 killable 版本。它用于获取信号量 sem。如果信号量不可用，它将被置为睡眠状态；如果向它传递了一个 fatal signals，则会将它从等待列表中删除，并且需要响应此信号。

4．down_trylock

```
int down_trylock(struct semaphore *sem)
{
    unsigned long flags;
    int count;

    spin_lock_irqsave(&sem->lock, flags);
    count = sem->count - 1;
    if (likely(count >= 0))
        sem->count = count;
    spin_unlock_irqrestore(&sem->lock, flags);

    return (count < 0);
}
```

down_trylock 函数尝试获得信号量 sem，如果能够立即获得，则获得信号量 sem 并返回 0；否则，返回非 0。它不会导致调用者睡眠，可以在中断上下文中使用。

5．down_ timeout

```
int down_timeout(struct semaphore *sem, long jiffies)
{
    unsigned long flags;
    int result = 0;

    spin_lock_irqsave(&sem->lock, flags);
    if (likely(sem->count > 0))
        sem->count--;
    else
        result = __down_timeout(sem, jiffies);
    spin_unlock_irqrestore(&sem->lock, flags);

    return result;
}
```

以下几个 down 函数最终都调用了__down_common 函数：

```
static noinline void __sched __down(struct semaphore *sem)
{
    __down_common(sem, TASK_UNINTERRUPTIBLE, MAX_SCHEDULE_TIMEOUT);
}
```

```
static noinline int __sched __down_interruptible(struct semaphore *sem)
{
    return __down_common(sem, TASK_INTERRUPTIBLE, MAX_SCHEDULE_TIMEOUT);
}

static noinline int __sched __down_killable(struct semaphore *sem)
{
    return __down_common(sem, TASK_KILLABLE, MAX_SCHEDULE_TIMEOUT);
}

static noinline int __sched __down_timeout(struct semaphore *sem, long jiffies)
{
    return __down_common(sem, TASK_UNINTERRUPTIBLE, jiffies);
}

static inline int __sched __down_common(struct semaphore *sem, long state,
                                        long timeout)
{
    struct task_struct *task = current;
    struct semaphore_waiter waiter;

    list_add_tail(&waiter.list, &sem->wait_list);
    waiter.task = task;
    waiter.up = 0;

    for (;;) {
        if (signal_pending_state(state, task))
            goto interrupted;
        if (timeout <= 0)
            goto timed_out;
        __set_task_state(task, state);
        spin_unlock_irq(&sem->lock);
        timeout = schedule_timeout(timeout);
        spin_lock_irq(&sem->lock);
        if (waiter.up)
            return 0;
    }

 timed_out:
    list_del(&waiter.list);
    return -ETIME;

 interrupted:
    list_del(&waiter.list);
    return -EINTR;
}
```

__down_common 首先进行状态检查和时间片检查；然后将当前进程的状态设置为 UNINTERRUPTIBLE；最后进行调度，并将当前进程阻塞。

Linux 的 V 函数是 up。当调用 up 之后，调用者不再拥有该信号量。函数实现代码如下：

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

```
void up(struct semaphore *sem)
{
    unsigned long flags;

    spin_lock_irqsave(&sem->lock, flags);
    if (likely(list_empty(&sem->wait_list)))
        sem->count++;
    else
        __up(sem);
    spin_unlock_irqrestore(&sem->lock, flags);
}

static noinline void __sched __up(struct semaphore *sem)
{
    struct semaphore_waiter *waiter = list_first_entry(&sem->wait_list,
        struct semaphore_waiter, list);
    list_del(&waiter->list);
    waiter->up = 1;
    wake_up_process(waiter->task);
}
```

6.3.3 信号量的使用

信号量的使用通常有以下 4 个步骤：

- DECLARE_MUTEX(sem);
- down(&sem); //获得信号量
- ...[CODE]... //临界区(被保护的资源)
- up(&sem); //释放信号量

与信号量相关的内核 API 如表 6.1 所示。

表 6.1 信号量的 API

函 数 原 型	功 能
DECLARE_MUTEX(name)	声明一个信号量 name 并初始化它的值为 0，即声明一个互斥锁
DECLARE_MUTEX_LOCKED(name)	声明一个互斥锁 name，把它的初始值配置为 0，即锁在创建时处在已锁状态
sema_init (struct semaphore *sem, int val)	初始化配置信号量的初值
init_MUTEX (struct semaphore *sem)	初始化一个互斥锁，把信号量 sem 的值设置为 1
init_MUTEX_LOCKED (struct semaphore *sem)	初始化一个互斥锁，把信号量 sem 的值设置为 0
down(_*)(struct semaphore * sem)	获得信号量 sem
up(struct semaphore * sem)	释放信号量 sem，即把 sem 的值加 1，假如 sem 的值为非正数，表明有任务等待该信号量，因此唤醒这些等待者

除了前面提到的信号量外，Linux 内核还提供了读/写信号量。读/写信号量对访问者进行了细分，或者为读者，或者为写者。读者在保持读/写信号量期间只能对该读/写信号量保护的共享资源进行读访问。如果一个任务除了需要读，可能还需要写，那么它必

华清远见 HQYJ.COM 华清远见教育集团官网：www.hqyj.com

须被归类为写者。在对共享资源访问之前必须先获得写者身份，当写者发现自己不需要写访问时，可以降级为读者身份。读/写信号量同时拥有的读者数不受限制，也就是说可以有任意多个读者同时拥有一个读/写信号量。它适于在读多写少的情况，在 Linux 内核中对进程的内存映像描述结构的访问就使用了读/写信号量进行保护。

一般来说，当对低开销、短期、中断上下文加锁时，优先考虑自旋锁；当对长期、持有锁需要休眠的任务时，可以优先考虑信号量；其他情况建议使用读/写信号量。读/写信号量的相关 API 如表 6.2 所示。

表 6.2 读/写信号量的 API

函数原型	功能
DECLARE_RWSEM(name)	声明一个读/写信号量 name 并对其进行初始化
init_rwsem(struct rw_semaphore *sem)	对读/写信号量 sem 进行初始化
down_read(struct rw_semaphore *sem)	读者调用该函数来得到读/写信号量 sem。该函数会导致调用者睡眠，因此只能在进程上下文中使用
down_read_trylock(struct rw_semaphore *sem)	该函数类似于 down_read，只是它不会导致调用者睡眠
down_write(struct rw_semaphore *sem)	写者使用该函数来得到读/写信号量 sem，它也会导致调用者睡眠，因此只能在进程上下文中使用
down_write_trylock(struct rw_semaphore *sem)	该函数类似于 down_write，只是它不会导致调用者睡眠
up_read(struct rw_semaphore *sem)	读者使用该函数释放读/写信号量 sem。它和 down_read 或 down_read_trylock 配对使用
up_write(struct rw_semaphore *sem)	写者调用该函数释放信号量 sem。它和 down_write 或 down_write_trylock 配对使用
downgrade_write(struct rw_semaphore *sem)	该函数用于把写者降级为读者

通过学习前面的内容，可以掌握信号量的原理及 API，下面的代码是内核模块中对信号量机制的使用。

```
#include<linux/init.h>
#include<linux/module.h>
#include<linux/sched.h>
#include<linux/sem.h>
MODULE_LICENSE("Dual BSD/GPL");

int num[2][5]={
{0,2,4,6,8},
    {1,3,5,7,9}
};
struct semaphore sem_one;
struct semaphore sem_two;
int thread_show_one(void *);
int thread_show_two(void *);
```


华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

```
int thread_show_one(void *p)
{
    int i;
    int *num=(int *)p;
    for(i=0;i<5;i++) {
        down(&sem_one);
        printk(KERN_INFO" Semaphore:%d\n",num[i]);
        up(&sem_two);
    }
    return 0;
}
int thread_show_two(void *p)
{
    int i;
    int *num=(int *)p;
    for(i=0;i<5;i++) {
        down(&sem_two);
        printk(KERN_INFO" Semaphore:%d\n",num[i]);
        up(&sem_one);
    }
    return 0;
}
static int semdemo _init(void)
{
    init_MUTEX(&sem_one);
    init_MUTEX(&sem_two);
    kernel_thread(thread_show_one,num[0],CLONE_KERNEL);
    kernel_thread(thread_show_two,num[1],CLONE_KERNEL);
    return 0;
}
static void semdemo _exit(void)
{
    printk(KERN_ALERT" semdemo module quit\n");
}
module_init(semdemo_init);
module_exit(semdemo _exit);
```

6.4 共享内存



6.4.1 什么是共享内存

共享内存是最快的 IPC 形式。两个不同进程 A、B 共享内存的意思是，同一块物理内存被映射到进程 A、B 各自的进程地址空间。进程 A 可以即时看到进程 B 对共享内存

中数据的更新，反之亦然。由于多个进程共享同一块内存区域，必然需要某种同步机制，互斥锁和信号量都可以。

采用共享内存通信的一个显而易见的好处是效率高，因为进程可以直接读/写内存，而不需要任何数据的复制。对于像管道和消息队列等通信方式，则需要在内核和用户空间进行 4 次复制数据，而共享内存则只复制两次数据：一次是从输入文件到共享内存区，另一次是从共享内存区到输出文件。实际上，进程之间在共享内存时，并不总是读/写少量数据后就解除映射，有新的通信时，会重新建立共享内存区域。而且是保持共享区域，直到通信完毕为止。这样，数据内容一直保存在共享内存中，并没有写回文件。共享内存中的内容往往是在解除映射时才写回文件的。因此，采用共享内存的通信方式效率是非常高的。

在 Linux 操作系统中，内核做两件事：一件是在内存中规划出一块区域来作为共享区；另一件就是把这个区域映射到参与通信的各个进程空间。具体做法是把一个已打开的文件所占用的内存空间作为共享区，然后通过调用 mmap()把这块区域映射到各个进程地址空间，从而使用户进程都可以看到这个共享区域。

mmap 原型如下：

```
void *mmap(void * start, size_t len, int prot, int flags, int fd, off_t offset);
```

其中，参数 fd 用来指定被映射的文件；offset 指定映射的起始位置偏移量；len 指定文件被映射部分的长度；start 用来指定映射到虚拟地址空间的起始地址。

6.4.2 共享内存的内核实现

通常，一个共享内存区由多个共享段组成，用来描述一个共享内存段的内核数据结构是 shmid_kernel，这个结构在 include\linux\shm.h 中定义：

```
struct shmid_kernel /* private to the kernel */
{
    struct kern_ipc_perm  shm_perm;
    struct file *     shm_file;
    unsigned long     shm_nattch;
    unsigned long     shm_segsz;
    time_t                shm_atim;
    time_t                shm_dtim;
    time_t                shm_ctim;
    pid_t                 shm_cprid;
    pid_t                 shm_lprid;
    struct user_struct    *mlock_user;
};
```

SHMID_KERNEL 数据结构成员说明如表 6.3 所示。

华清远见

华清远见 HQYJ.COM

华清远见教育集团官网：www.hqyj.com

表 6.3　SHMID_KERNEL 数据结构成员说明

成　　员	描　　述
shm_perm	描述进程间通信许可的结构
shm_file	指向共享内存页交换文件对象指针
shm_nattch	挂接到本段共享内存的进程数
shm_segsz	段大小
shm_atim	最后挂接时间
shm_dtim	最后解除挂接时间
shm_ctim	最后变化时间
shm_cprid	创建进程的 PID
shm_lprid	最后使用进程的 PID
*mlock_user	指向上锁的用户

6.4.3　共享内存的使用

内核为共享内存机制提供了 4 种操作：SHMGET、SHMAT、SHMDT 和 SHMCTL，它们各自对应库函数 shmget()、shmat()、shmdt()和 shmctl()，其系统调用分别由 sys_shmget()、sys_shmat()、sys_shmdt()、sys_shmctl()实现。

进行应用程序编程时，需要包含下面几个头文件：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

1．共享内存的打开和创建

shmget()用来打开或创建一个共享内存区。这个函数的内部是通过系统调用 SYSCALL_DEFINE3 实现的。代码如下（见 linux 源代码目录下 ipc\shm.c）：

```
SYSCALL_DEFINE3(shmget, key_t, key, size_t, size, int, shmflg)
{
	struct ipc_namespace *ns;
	struct ipc_ops shm_ops;
	struct ipc_params shm_params;

	ns = current->nsproxy->ipc_ns;

	shm_ops.getnew = newseg;
	shm_ops.associate = shm_security;
	shm_ops.more_checks = shm_more_checks;

	shm_params.key = key;
	shm_params.flg = shmflg;
	shm_params.u.size = size;
```


华清远见

```
        return ipcget(ns, &shm_ids(ns), &shm_ops, &shm_params);
    }
```

参数 key 是用户给定的键值。参数 shmflg 是该函数的功能标志。如果 shmflg 的 IPC_CREATE=1，则这个系统调用会为用户创建或打开一个共享内存区，并返回其标识号；如果 IPC_CREATE=0，则会在系统已有的共享内存区中寻找与键值相同的共享内存区，找到后返回它的标识号并打开它。

这里用到了两个结构：ipc_params 和 ipc_ops，其实现都在 ipc\util.h 中。代码如下：

```
struct ipc_params {
    key_t key;
    int flg;
    union {
        size_t size;  /* for shared memories */
        int nsems;    /* for semaphores */
    } u;              /* holds the getnew() specific param */
};

struct ipc_ops {
    int (*getnew) (struct ipc_namespace *, struct ipc_params *);
    int (*associate) (struct kern_ipc_perm *, int);
    int (*more_checks) (struct kern_ipc_perm *, struct ipc_params *);
};
```

2. 共享内存与进程的链接

如果一个进程已创建或打开了一个共享内存，则在需要使用它时，要调用函数 shmat()把该共享内存链接到进程上，即要把待使用的共享内存映射到进程空间。函数 shmat()通过系统调用 SYSCALL_DEFINE3()实现，代码如下（见 ipc\shm.c）：

```
SYSCALL_DEFINE3(shmat, int, shmid, char __user *, shmaddr, int, shmflg)
{
    unsigned long ret;
    long err;

    err = do_shmat(shmid, shmaddr, shmflg, &ret);
    if (err)
        return err;
    force_successful_syscall_return();
    return (long)ret;
}
```

其中，shmid 是共享内存的标识；参数 shmaddr 是映射地址，如果该地址为 0，则由内核决定；参数 shmflg 为共享内存的标识，如果 shmflg 的值为 SHM_RDONLY，则进程以只读的方式访问共享内存，否则，以读/写的方式访问共享内存。

3. 断开共享内存与进程的链接

调用函数 shmdt()可以断开共享内存与进程的链接。该函数是由系统调用 SYSCALL_DEFINE1 实现的，代码如下：

华清远见 HQYJ.COM 华清远见教育集团官网：www.hqyj.com




```
SYSCALL_DEFINE1(shmdt, char __user *, shmaddr){
     struct mm_struct *mm = current->mm;
     struct vm_area_struct *vma, *next;
     unsigned long addr = (unsigned long)shmaddr;
     loff_t size = 0;
     int retval = -EINVAL;
     if (addr & ~PAGE_MASK)
          return retval;
     down_write(&mm->mmap_sem);
     vma = find_vma(mm, addr);
     while (vma) {
          next = vma->vm_next;
          if ((vma->vm_ops == &shm_vm_ops) &&
               (vma->vm_start - addr)/PAGE_SIZE == vma->vm_pgoff) {


               size = vma->vm_file->f_path.dentry->d_inode->i_size;
               do_munmap(mm, vma->vm_start, vma->vm_end - vma->vm_start);
               retval = 0;
               vma = next;
               break;
          }
          vma = next;
     }
     size = PAGE_ALIGN(size);
     while (vma && (loff_t)(vma->vm_end - addr) <= size) {
          next = vma->vm_next;
          /* finding a matching vma now does not alter retval */
          if ((vma->vm_ops == &shm_vm_ops) &&
               (vma->vm_start - addr)/PAGE_SIZE == vma->vm_pgoff)
               do_munmap(mm, vma->vm_start, vma->vm_end - vma->vm_start);
          vma = next;
     }

     up_write(&mm->mmap_sem);
     return retval;
}
```

唯一的一个参数 shmaddr 是链接地址。

6.5 消息队列



6.5.1 什么是消息队列

消息队列是系统定义的内存块，用于临时存储消息。消息队列就是一个消息的链表。可以把消息看做一个记录，具有特定的格式及特定的优先级。对消息队列有写权限的进程可以按照一定的规则为其添加新消息；对消息队列有读权限的进程则可以从消息队列

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

中读取消息。目前，主要有两种类型的消息队列：POSIX 消息队列和系统 V 消息队列，系统 V 消息队列目前被大量使用。考虑到程序的可移植性，新开发的应用程序应尽量使用 POSIX 消息队列。

消息队列的工作机制如图 6.2 所示。

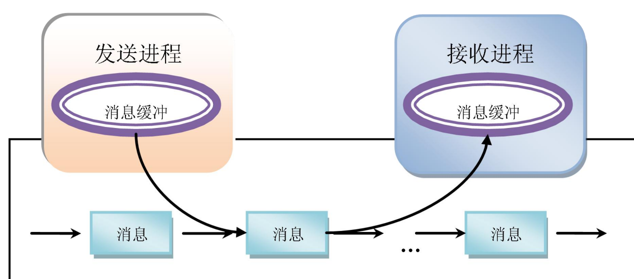


图 6.2　消息队列的工作机制

6.5.2　消息队列的内核实现

消息队列主要用来实现消息传递，因此我们需要掌握消息及消息队列的结构。内核分别用 msgbuf、msg_msgseg、msg_queue 描述用户空间的消息缓冲区、内核空间的消息结构和消息队列结构。

对于开发人员来说，用户空间的每个消息都类似如下数据结构：

```
struct msgbuf
{
    long mtype;
    char mtext[1];
};
```

mtype 成员代表消息类型，从消息队列中读取消息的一个重要依据就是消息的类型；mtext 是消息内容，当然长度不一定为 1。因此，对于发送消息来说，首先预置一个 msgbuf 缓冲区并写入消息类型和内容，调用相应的发送函数即可；对于读取消息来说，首先分配这样一个 msgbuf 缓冲区，然后把消息读入该缓冲区即可。

msg_msgseg 结构的定义在 ipc\msgutil.c 文件中，代码如下：

```
struct msg_msgseg {
    struct msg_msgseg* next;
    /* the next part of the message follows immediately */
};
```

一个大型消息的结构图如图 6.3 所示。

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

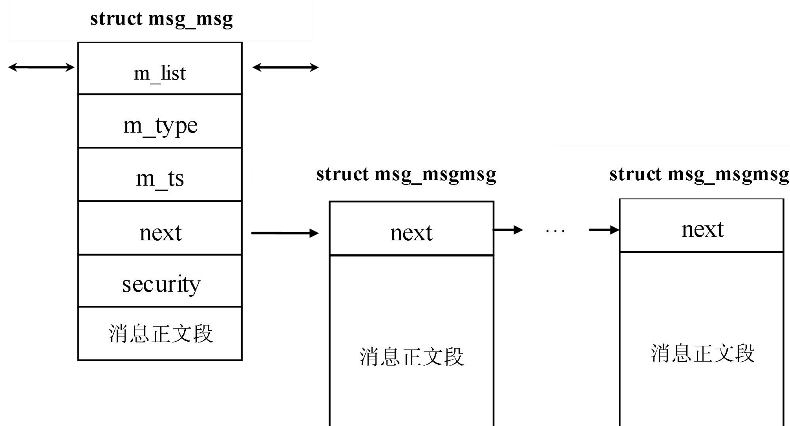


图 6.3　大型消息的结构图

其中，msg_msg 是一个稍微复杂的结构。为什么会设计这样一个结构呢？这是出于对消息的维护和管理的考虑。msg_msg 称为内核空间的首页结构，而 msg_msgmsg 称为一般页结构。msg_msg 的定义如下：

```
/* one msg_msg structure for each message */
struct msg_msg {
     struct list_head m_list;
     long  m_type;
     int m_ts;          /* message text size */
     struct msg_msgseg* next;
     void *security;
     /* the actual message follows immediately */
};
```

其中，m_list 表示链表结构；m_type 是消息类型；m_ts 是消息文本大小；next 表示下一个消息片段页。

另一个重要的结构是消息队列结构 msg_queue。每个消息队列都有一个 msg_queue 结构类型的队列头，msg_queue 的定义如下：

```
/* one msq_queue structure for each present queue on the system */
struct msg_queue {
     struct kern_ipc_perm q_perm;
     time_t q_stime;            /* last msgsnd time */
     time_t q_rtime;            /* last msgrcv time */
     time_t q_ctime;            /* last change time */
     unsigned long q_cbytes;    /* current number of bytes on queue */
     unsigned long q_qnum; /* number of messages in queue */
     unsigned long q_qbytes;    /* max number of bytes on queue */
     pid_t q_lspid;             /* pid of last msgsnd */
     pid_t q_lrpid;             /* last receive pid */

     struct list_head q_messages;
     struct list_head q_receivers;
```


华清远见 HQYJ.COM　　华清远见教育集团官网：www.hqyj.com

```
        struct list_head q_senders;
};
```

其中，q_messages 是消息队列；q_receivers 是接收信号进程等待队列；q_senders 是发送信号进程等待队列。这些队列都是链表结构。

6.5.3 消息队列的使用

消息队列的内核持续性要求每个消息队列都在系统范围内对应唯一的键值，所以，要获得一个消息队列的描述符，只需提供该消息队列的键值即可。消息队列的主要调用有以下 4 个。

- msgget：调用者提供一个消息队列的键值，当这个消息队列存在时，这个消息调用负责返回这个队列的标识号；如果这个队列不存在，就创建一个消息队列，然后返回这个消息队列的标识号，由 sys_msgget 执行。
- msgsnd：向一个消息队列发送一个消息，由 sys_msgsnd 执行。
- msgrcv：从一个消息队列中收到一个消息，由 sys_msgrcv 执行。
- msgctl：在消息队列上执行指定的操作。根据参数的不同和权限的不同，可以执行检索、删除等操作，由 sys_msgctl 执行。

这几个函数在使用时需要包括以下 3 个头文件：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
```

进程创建一个消息队列的函数是 msgget()，其对应的系统调用是 sys_msgget()，实现代码如下（见 ipc\msg.c）：

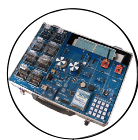
```
SYSCALL_DEFINE2(ms gget, key_t, key, int, msgflg){
        struct ipc_namespace *ns;
        struct ipc_ops msg_ops;
        struct ipc_params msg_params;
        ns = current->nsproxy->ipc_ns;
        msg_ops.getnew = newque;
        msg_ops.associate = msg_security;
        msg_ops.more_checks = NULL;
        msg_params.key = key;
        msg_params.flg = msgflg;
        return ipcget(ns, &msg_ids(ns), &msg_ops, &msg_params);
}
```

其中，参数 key 是用户给定的键值。如果 key=0，系统会为进程创建一个进程自用的消息队列；否则，创建或打开一个消息队列，msgflg 是该函数的功能标识。

类似的，消息的发送和接收分别是 msgsnd 和 msgrcv，所对应的系统调用是 sys_msgsnd 和 sys_msgrcv，其实现代码如下（见 ipc\msg.c）：

```
asmlinkage long
sys_msgsnd(int msqid, struct msgbuf __user *msgp, size_t msgsz, int msgflg)
{
```


华清远见 HQYJ.COM 华清远见教育集团官网：www.hqyj.com




```
        long mtype;

        if (get_user(mtype, &msgp->mtype))
            return -EFAULT;
        return do_msgsnd(msqid, mtype, msgp->mtext, msgsz, msgflg);
    }

    asmlinkage long sys_msgrcv(int msqid, struct msgbuf __user *msgp, size_t msgsz,long
msgtyp, int msgflg)
    {
        long err, mtype;

        err =  do_msgrcv(msqid, &mtype, msgp->mtext, msgsz, msgtyp, msgflg);
        if (err < 0)
            goto out;

        if (put_user(mtype, &msgp->mtype))
            err = -EFAULT;
    out:
        return err;
    }
```

这两个函数分别调用了 do_msgsnd 和 do_msgrcv，程序代码留给读者自行研究。消息队列应用相对较简单，下面的实例基本上覆盖了对消息队列的所有操作，包括创建、发送、读取、改变权限及删除消息队列等。

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <ctype.h>
    #include <string.h>
    #include <sys/types.h>
    #include <sys/ipc.h>
    #include <sys/msg.h>
    #define MAX_SEND_SIZE 256
    struct mymsgbuf {
    long mtype;
    char mtext[MAX_SEND_SIZE];
    };
    void send_message(int qid, struct mymsgbuf *qbuf, long type, char *text);
    void read_message(int qid, struct mymsgbuf *qbuf, long type);
    void remove_queue(int qid);
    void change_queue_mode(int qid, char *mode);
    void usage(void);
    int main(int argc, char *argv[])
    {
        key_t key;
        int msgqueue_id;
        struct mymsgbuf qbuf;
        if(argc == 1)
        usage();
        key = ftok(".", 'm'); /*创建 IPC 标识符，需要先获得一个 key */
        if((msgqueue_id = msgget(key, IPC_CREAT|0660)) == -1)
    /*如果 msg_id 不存在，则创建；否则，返回已经存在的队列标识符*/
```


华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

```
        {
            perror("msgget");
            exit(1);
        }
        switch(tolower(argv[1][0]))
        {
        case 's':
            send_message(msgqueue_id, (struct mymsgbuf *)&qbuf,atol(argv[2]),
            argv[3]);
            break;
        case 'r':
            read_message(msgqueue_id, &qbuf, atol(argv[2]));
            break;
        case 'd':
            remove_queue(msgqueue_id);
            break;
        case 'm':
            change_queue_mode(msgqueue_id, argv[2]);
            break;
        default: usage();
        }
        return(0);
}
void send_message(int qid, struct mymsgbuf *qbuf, long type, char *text)
{
        printf("正在发送消息 …n");
        qbuf->mtype = type;
        strcpy(qbuf->mtext, text);
        if((msgsnd(qid, (struct msgbuf *)qbuf,strlen(qbuf->mtext)+1, 0)) ==-1)
/* 向队列发生消息 */
        {
            perror("msgsnd");
            exit(1);
        }
}
void read_message(int qid, struct mymsgbuf *qbuf, long type)
{
        printf("读取消息中…n");
        qbuf->mtype = type;
        /* 读取来自队列的消息*/
        msgrcv(qid, (struct msgbuf *)qbuf, MAX_SEND_SIZE, type, 0);
        printf("Type: %ld Text: %sn", qbuf->mtype, qbuf->mtext);
}
void remove_queue(int qid)
{
        msgctl(qid, IPC_RMID, 0); /* Remove the queue */
}
void change_queue_mode(int qid, char *mode)
{
        struct msqid_ds myqueue_ds;
        msgctl(qid, IPC_STAT, &myqueue_ds); /*获得消息队列信息*/
        /* Convert and load the mode */
        sscanf(mode, "%ho", &myqueue_ds.msg_perm.mode);
```

```
    msgctl(qid, IPC_SET, &myqueue_ds); /* Update the mode */
}
void usage(void)
{
    fprintf(stderr, "msgtool - A utility for tinkering with msg queuesn");
    fprintf(stderr, "nUSAGE: msgtool (s)end n");
    fprintf(stderr, " (r)ecv n");
    fprintf(stderr, " (d)eleten");
    fprintf(stderr, " (m)ode n");
    exit(1);
}
```

6.6 管道



6.6.1 什么是管道

管道是 Linux 从 UNIX 系统继承过来的 IPC 机制，也是 UNIX 早期的一个重要通信机制。其思想是在内存中创建一个共享文件，从而使通信双方利用这个文件进行信息传递。因为这种方式具有单向传送的特点，所以这个作为传递信息的共享文件就称为管道。管道是半双工的，数据只能向一个方向流动；需要双方通信时，需要建立起两个管道。管道对于管道两端的进程而言，就是一个文件，但它不是普通的文件，它不属于某种文件系统，而是单独构成一种文件系统，并且只存在于内存中。

如图 6.4 所示为两个进程通过管道进程通信的示意图。显然，如果两个或多个进程同时对一个进程进行读/写，那么这些进程必须使用锁机制或信号量机制对其进行同步。当管道空闲时，在有数据写入之前，读进程一直被阻塞；同样，当管道满时，在其中数据被读出之前，写进程将发生阻塞。



图 6.4　管道原理示意图

在管道的实现中，根据通信使用的文件是否具有名称，可分为匿名管道和命名管道两种。匿名管道没有名字，所以只能提供给进程家族中的父子进程间通信使用。命名管道又称 FIFO（先进先出），是一个能在互不相关进程之间传送数据的特殊文件。它是在实际文件系统的基础上实现的一种通信机制。

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

6.6.2　管道的内核实现

管道是内核为需要进行通信的进程之间铺设的一个共享物理页，用来描述这个物理页属性的数据结构是 pipe_inode_info（见 include\linux\pipe_fs_i.h），代码如下：

```
struct pipe_inode_info {
	wait_queue_head_t wait;
	unsigned int nrbufs, curbuf;
	struct page *tmp_page;
	unsigned int readers;
	unsigned int writers;
	unsigned int waiting_writers;
	unsigned int r_counter;
	unsigned int w_counter;
	struct fasync_struct *fasync_readers;
	struct fasync_struct *fasync_writers;
	struct inode *inode;
	struct pipe_buffer bufs[PIPE_BUFFERS];
};
```

pipe_inode_info 数据结构成员说明如表 6.4 所示。

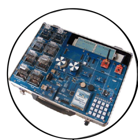
表 6.4　pipe_inode_info 数据结构成员说明

成　　员	描　　述
wait	描述管道的等待队列
nrbufs, curbuf	包含待读数据的缓冲区数和包含待读数据的第一个缓冲区的索引
tmp_page	高速缓存页框指针
readers	读进程的编号
writers	写进程的编号
waiting_writers	被阻塞的写进程计数器
r_counter	以只读方式访问管道的进程计数器
w_counter	以只写方式访问管道的进程计数器
fasync_readers	用于通过信号进行读的异步 I/O 通知
fasync_writers	用于通过信号进行写的异步 I/O 通知
Inode	管道文件的节点
bufs[PIPE_BUFFERS]	缓冲区数组

其中，bufs[PIPE_BUFFERS]是构成管道的内存缓冲区，该缓冲区通过 pipe_buffer 结构进行描述，代码如下：

```
struct pipe_buffer {
	struct page *page;
	unsigned int offset, len;
	const struct pipe_buf_operations *ops;
	unsigned int flags;
	unsigned long private;
```


华清远见 HQYJ.COM　华清远见教育集团官网：www.hqyj.com




```
};
```

可以看出，管道实质上是一个被当做文件来管理的内存缓冲区。结构中的 page 是管道缓冲区页框的描述符地址，offset 是页框内有效数据的当前位置，ops 是缓冲区的操作函数指针，结构如下：

```
struct pipe_buf_operations {
    int can_merge;
    void * (*map)(struct pipe_inode_info *, struct pipe_buffer *, int);
    void (*unmap)(struct pipe_inode_info *, struct pipe_buffer *, void *);
    int (*confirm)(struct pipe_inode_info *, struct pipe_buffer *);
    void (*release)(struct pipe_inode_info *, struct pipe_buffer *);
    int (*steal)(struct pipe_inode_info *, struct pipe_buffer *);
    void (*get)(struct pipe_inode_info *, struct pipe_buffer *);
};
```

6.6.3 管道的读/写规则

管道两端可分别用描述字 fd[0]和 fd[1]来描述，需要注意的是，管道的两端是固定了任务的。即一端只能用于读，由描述字 fd[0]表示，称为管道读端；另一端则只能用于写，由描述字 fd[1]来表示，称为管道写端。如果试图从管道写端读取数据或者向管道读端写入数据，都将导致错误发生。一般文件的 I/O 函数都可以用于管道，如 close、read、write 等。

华清远见

1．从管道中读取数据

如果管道的写端不存在，则认为已经读到了数据的末尾，读函数返回的读出字节数为 0。

当管道的写端存在时，如果请求的字节数大于 PIPE_BUF，则返回管道中现有的数据字节数；如果请求的字节数不大于 PIPE_BUF，则返回管道中现有的数据量小于请求的数据量字节数；或者返回请求的字节数。

2．向管道中写入数据

向管道中写入数据时，Linux 将不保证写入的原子性，管道缓冲区一有空闲区域，写进程就会试图向管道写入数据。如果读进程不读出管道缓冲区中的数据，那么写操作将一直阻塞。

6.7 本章习题



1．和进程管理相关的内核文件都有哪些？

2．什么是进程和线程？

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

3．什么是进程描述符？怎样得到当前进程的进程描述符？
4．进程的内核栈有多大？
5．进程的状态都有哪些？在什么情况下发生转化？
6．Linux 中所有进程之间的关系是怎么样的？
7．什么是命名管道和匿名管道？它们的区别是什么？

华清远见 HQYJ.COM

第 7 章 中断与系统调用

中断，是指外部事件发生以后，处理器中止当前程序，转而执行另一个程序的过程。与普通程序转移方式不同，其转移目标地址不是由指令提供，而是由硬件提供的。中断也是外设与系统进行通信的手段。本章主要介绍 Linux 操作系统的中断与其处理过程，以及系统调用的实现。

华清远见 HQYJ.COM

华清远见教育集团官网：www.hqyj.com

7.1 什么是中断



Linux 系统支持很多不同种类的硬件设备。应用程序可以以同步方式读/写，控制这些设备，也就是说应用可以发出一个设备请求，然后等待，一直等到设备完成操作以后才返回。但这种方法的效率却非常低，因为内核需要花费大量的时间轮寻设备是否完成操作。一个更为有效的方法是，内核发出请求以后，操作系统继续其他进程的工作，等设备完成操作以后，给操作系统发送一个中断，操作系统再继续处理和此设备有关的操作。

在将多个设备的中断信号送往 CPU 的中断插脚之前，系统经常使用中断控制器来综合多个设备的中断。这样既可以节约 CPU 的中断插脚，也可以提高系统设计的灵活性。中断控制器用来控制系统的中断，它包括屏蔽和状态寄存器。设置屏蔽寄存器的各个位可以允许或屏蔽某一个中断，状态寄存器则用来返回系统中正在使用的中断。

大多数处理器处理中断的过程都相同。当一个设备发出中段请求时，CPU 停止正在执行的指令，转而跳到包括中断处理代码或包括指向中断处理代码的转移指令所在的内存区域。这些代码一般在 CPU 的中断方式下运行。在此方式下，将不会再有中断发生。但有些 CPU 的中断有自己的优先权，这样，更高优先权的中断则可以发生。这意味着第一级的中断处理程序必须拥有自己的堆栈，以便在处理更高级别的中断前保存 CPU 的执行状态。当中断处理完毕以后，CPU 将恢复到以前的状态，继续执行中断处理前正在执行的指令。中断处理程序十分简单有效，这样，操作系统就不会花太长的时间屏蔽其他的中断。

华清远见

7.2 嵌入式平台硬件中断特点



硬件之所以会产生中断，是因为外设需要通知操作系统它发生了一些事情，但是中断的功能仅仅是一个设备产生事件，当事件发生时中断处理程序只知道有事情发生了，但具体发生了什么事情还要亲自到设备那里去看才行。也就是说，当中断处理程序得知设备发生了一个中断时，它并不知道设备发生了什么事情，只有当它访问了设备上的一些状态寄存器以后，才能知道具体发生了什么，要怎么去处理。

设备通过中断线向中断控制器发送高电平告诉操作系统它产生了一个中断，而操作系统会从中断控制器的状态位知道是哪条中断线上产生了中断。并不是每个设备都可以向中断线上发中断信号，只有对某一条确定的中断线拥有了控制权，才可以向这条中断线上发送信号。由于计算机的外部设备越来越多，所以中断线是非常宝贵的资源。要使

用中断线，就必须进行中断线的申请，就是IRQ（Interrupt Requirement），通常把申请一条中断线称为申请一个IRQ或申请一个中断号。

IRQ是非常宝贵的，所以只有当设备需要中断时才申请占用一个IRQ，或者在申请IRQ时采用共享中断的方式，这样可以让更多的设备使用中断。无论对IRQ的使用方式是独占还是共享，申请IRQ的过程都是一样的，分为以下3步：

（1）将所有的中断线探测一遍，看看哪些中断还没有被占用。从这些还没有被占用的中断中选一个作为该设备的IRQ。

（2）通过中断申请函数申请选定的IRQ，指定申请的方式是独占还是共享。

（3）根据中断申请函数的返回值决定是否重新申请或放弃申请并返回错误。

但是，目前在大多数嵌入式平台上，每个设备的中断号已经被固定分配好，不可再变，所以在这样的平台上为外部设备（外设）申请中断时，就不需要做第一步了，而是直接以分配好的中断号去申请中断。例如，在三星的S3c2410处理器中，它的各外部设备在中断控制器中所分配的情况如图7.1所示。

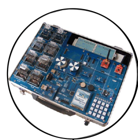
INTMSK	Bit	Description	Initial State
INT_ADC	[31]	0 = Service available, 1 = Masked	1
INT_RTC	[30]	0 = Service available, 1 = Masked	1
INT_SPI1	[29]	0 = Service available, 1 = Masked	1
INT_UART0	[28]	0 = Service available, 1 = Masked	1
INT_IIC	[27]	0 = Service available, 1 = Masked	1
INT_USBH	[26]	0 = Service available, 1 = Masked	1
INT_USBD	[25]	0 = Service available, 1 = Masked	1
Reserved	[24]	Not used	1
INT_UART1	[23]	0 = Service available, 1 = Masked	1
INT_SPI0	[22]	0 = Service available, 1 = Masked	1
INT_SDI	[21]	0 = Service available, 1 = Masked	1
INT_DMA3	[20]	0 = Service available, 1 = Masked	1
INT_DMA2	[19]	0 = Service available, 1 = Masked	1
INT_DMA1	[18]	0 = Service available, 1 = Masked	1
INT_DMA0	[17]	0 = Service available, 1 = Masked	1
INT_LCD	[16]	0 = Service available, 1 = Masked	1
INT_UART2	[15]	0 = Service available, 1 = Masked	1
INT_TIMER4	[14]	0 = Service available, 1 = Masked	1
INT_TIMER3	[13]	0 = Service available, 1 = Masked	1
INT_TIMER2	[12]	0 = Service available, 1 = Masked	1
INT_TIMER1	[11]	0 = Service available, 1 = Masked	1
INT_TIMER0	[10]	0 = Service available, 1 = Masked	1
INT_WDT	[9]	0 = Service available, 1 = Masked	1
INT_TICK	[8]	0 = Service available, 1 = Masked	1
nBATT_FLT	[7]	0 = Service available, 1 = Masked	1

图7.1　S3c2410平台外部设备在中断控制器所分配的情况

对应这种外设中断号固定的平台，在移植Linux内核时要注意修改include/asm/arch-xxxx/irqs.h文件中的外设中断号定义，使之与硬件手册规定的一致。例如，在include/asm-arm/arch-s3c2410/irqs.h文件中对外设的中断号定义如下：

```
/* Interrupt Controller */
```


华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com




```
#define IRQ_EINT0      0    /* External interrupt 0 */
#define IRQ_EINT1      1    /* External interrupt 1 */
#define IRQ_EINT2      2    /* External interrupt 2 */
#define IRQ_EINT3      3    /* External interrupt 3 */
#define IRQ_EINT4_7         4    /* External interrupt 4 ~ 7 */
#define IRQ_EINT8_23   5    /* External interrupt 8 ~ 23 */
#define IRQ_RESERVED6 6    /* Reserved for future use */
#define IRQ_BAT_FLT         7
#define IRQ_TICK       8    /* RTC time tick interrupt  */
#define IRQ_WDT             9    /* Watch-Dog timer interrupt */
#define IRQ_TIMER0          10   /* Timer 0 interrupt */
#define IRQ_TIMER1          11   /* Timer 1 interrupt */
#define IRQ_TIMER2          12   /* Timer 2 interrupt */
#define IRQ_TIMER3          13   /* Timer 3 interrupt */
#define IRQ_TIMER4          14   /* Timer 4 interrupt */
#define IRQ_UART2      15   /* UART 2 interrupt  */
#define IRQ_LCD             16   /* reserved for future use */
#define IRQ_DMA0       17   /* DMA channel 0 interrupt */
#define IRQ_DMA1       18   /* DMA channel 1 interrupt */
#define IRQ_DMA2       19   /* DMA channel 2 interrupt */
#define IRQ_DMA3       20   /* DMA channel 3 interrupt */
#define IRQ_SDI             21   /* SD Interface interrupt */
#define IRQ_SPI0       22   /* SPI interrupt */
#define IRQ_UART1      23   /* UART1 receive interrupt */
#define IRQ_RESERVED   24
#define IRQ_USBD       25   /* USB device interrupt */
#define IRQ_USBH       26   /* USB host interrupt */
#define IRQ_IIC             27   /* IIC interrupt */
#define IRQ_UART0      28   /* UART0 transmit interrupt */
#define IRQ_SPI1       29   /* UART1 transmit interrupt */
#define IRQ_RTC             30   /* RTC alarm interrupt */
#define IRQ_ADCTC      31   /* ADC EOC interrupt */
#define NORMAL_IRQ_OFFSET 32
```

读者可以自己比较一下，看看程序中的定义与硬件文档中的定义是否一致。

7.3 Linux 内核中断机制概述



Linux 系统的中断就是通常意义上的“中断处理程序”，它直接处理由硬件发过来的中断信号。当 Linux 内核收到中断请求后，它首先判断中断源，然后调用相应的设备驱动程序，驱动程序会去设备上查看其状态寄存器，以了解发生了什么事情，并进行相应的操作。

Linux 内核与中断相关的部分包括：硬中断、下半部任务和软中断几种。下面就简要地讨论上述几类内核任务的特点。

华清远见 HQYJ.COM

华清远见教育集团官网：www.hqyj.com

1. 硬中断

硬中断是指那些由处理器以外的外设产生的中断，这些中断被处理器接收后交给内核中的中断处理程序处理。要注意的是：第一，硬中断是异步产生的，中断发生后立刻得到处理，也就是说中断操作可以抢占内核中正在运行的代码。这一点非常重要。第二，中断操作是发生在中断上下文中的（所谓中断上下文，指的是和任何进程无关的上下文环境）。中断上下文中不可以使用进程相关的资源，也不能进行调度或睡眠。因为调度会引起睡眠，但睡眠必须针对进程而言（睡眠其实是标记进程状态，然后把当前进程推入睡眠列队），而异步发生的中断处理程序根本不知道当前进程的任何信息，也不关心当前哪个进程在运行，它完全是一个过客。

2. 下半部任务

下半部任务的由来完全出自上面提到的硬中断的影响。硬中断任务（处理程序）是一个快速、异步、简单地对硬件做出迅速响应并在最短时间内完成必要操作的中断处理程序。硬中断处理程序可以抢占内核任务并且执行时还会屏蔽同级中断或其他中断，因此，中断处理必须要快、不能阻塞。这样，对于一些要求处理过程比较复杂的任务就不适合在中断任务中一次处理。例如，在网卡接收数据的过程中，首先网卡发送中断信号通知 CPU 获取数据，然后系统从网卡中读取数据存入系统缓冲区中，接下来解析数据然后送入应用层。如果这些都让中断处理程序来处理，显然过程太长，造成新来的中断丢失。因此 Linux 将这种任务分割为两个部分，中断处理程序短平快地处理与硬件相关的操作（如从网卡读数据到系统缓存），而把对时间要求相对宽松的任务（如解析数据的工作）放在另一个部分执行，这个部分就是下半部任务。下半部任务是一种推后执行任务，它将某些不是很紧迫的任务推迟到系统更方便的时刻运行。内核中实现下半部的手段经过不断演化，目前已经从最原始的 BH（Bottom Half）衍生出 tasklet、软中断（Softirq）、工作队列（Work Queues）。

3. 软中断

软中断不像硬中断那样是由硬件中断信号触发执行的，所以也不同于硬件中断那样随时都能够被执行。总的来讲，软中断会在内核处理任务完毕后返回用户级程序前得到处理机会。具体来讲，有 3 个时刻它将会执行 do_softirq 函数：这 3 个时刻分别为硬件中断操作完成后、系统调用返回时、内核调度程序中。从中可以看出软中断会紧随硬中断处理，所以抢占内核任务至少在时钟中断后总有机会运行一次。另外，软中断可以在不同处理器上并发执行。软中断结构定义及与其相关操作的定义如下：

```
struct softirq_action
{
    void (*action)(struct softirq_action *);
    void *data;
};
asmlinkage void do_softirq(void);
extern void open_softirq(int nr, void (*action)(struct softirq_action*), void
```


华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

```
*data);
    extern void softirq_init(void);
    #define __cpu_raise_softirq(cpu, nr) do { softirq_pending(cpu) |= 1UL << (nr); } while (0)
    extern void FASTCALL(cpu_raise_softirq(unsigned int cpu, unsigned int nr));
    extern void FASTCALL(raise_softirq(unsigned int nr));
```

使用时先执行 open_softirq，需要触发此软中断时使用 raise_softirq 或 cpu_raise_softirq。

在有对称多处理器的机器上，两个任务就可以真正地在临界区中同时执行了，这种类型称为真并发。相对而言，在单处理器上并发其实并不是真的同时发生，而是相互交错执行，是伪并发。但它们都同样会造成竞争条件，而且也需要同样的保护。

软中断属于底层的机制，一般除了在网络子系统和 SCSI 子系统这样对性能要求很高或要求并发处理时，才会选择使用软中断。软中断虽然灵活性高和效率高，但是必须处理复杂的同步处理（因为它可在多处理器上并发），所以通常都不直接使用，而是作为支持 tasklet 和 bottom half 的基础。

需要说明的是，软中断的执行也处于中断上下文中，所以中断上下文对它的限制是和硬中断一样的，都不能进入阻塞状态。

tasklet 和 bottom half 都是建立在软中断之上的两种延迟机制，其具体不同之处在于软中断是静态分配的，而且同类软中断可以并发地在几个 CPU 上运行。tasklet 可以动态分配，并且不同种类的 Tasklets 可以并发地在几个 CPU 上运行，但同类的 tasklets 不可以。Bottom half 只能静态分配。实际上，下半部分是一个不能与其他下半部分并发执行的高优先级 tasklet，即使它们类型不同，而且在不同 CPU 上运行。tasklet 可以理解为软中断的派生，所以它的调度时机与软中断一致。内核中对 tasklets 的解释和定义如下：

```
    struct tasklet_struct
    {
        struct tasklet_struct *next;
        unsigned long state;
        atomic_t count;
        void (*func)(unsigned long);
        unsigned long data;
    };
```

可见，tasklets 与 softirq 的主要区别就是，在同一时刻只能用一个 CPU 来运行一个 tasklet。而 softirq 则不然，可以在不同 CPU 上运行同一个 softirq，但要注意做好相关的保护工作。而 tasklets 与 BH 的区别是不同的 tasklets 可以在同一时刻运行在不同的 CPU 上，而 BH 是不可以的。而在 tasklets 的结构定义中，最重要的就是 func 成员，tasklets 所指的地址就是最终要执行的处理函数。

内核中需要延迟执行的多数任务都可以利用 tasklet 来完成，由于同类 tasklet 本身已经进行了同步保护，所以使用 tasklet 相比软中断要简单得多，而且效率也很高。

Bottom Half 是 Linux 最早的内核延迟方法，它结构简单且容易控制，因为所有的 BH 处理程序都被严格地顺序执行，不允许任何两个 BH 处理程序同时并发执行，即使它们的类型不同也不可以，这样一来，BH 执行期间减少了许多同步保护。但是 BH 不

得不被淘汰，因为它的“简便”牺牲了多处理器并发处理的高性能，就像一队人过独木桥那样，速度受到限制。

任务列队是 BH 的替代品，来自 BH，所以它的属性也和 BH 相同。它的原意在于简化 BH 的操作接口，但它的随意性（数量随意、执行时机随意）却给系统带来了混乱，所以现在已经被工作队列所取代。

7.4 编写中断处理程序 ISR



7.4.1 中断处理系统结构

Linux 中断处理子系统的基本任务是将中断正确路由到中断处理代码中的正确位置。这些代码必须了解系统的中断拓扑结构。Linux 使用一组指针来指向包含处理系统中断的例程的调用地址。这些例程属于对应于此设备的设备驱动，同时由它负责在设备初始化时为每个设备驱动申请其请求的中断。要了解内核对中断的组织，首先需要熟悉几个重要的结构类型，第一个就是 irqaction，它定义在 include/linux/interrupt.h 中，代码如下：

```
struct irqaction {
    void (*handler)(int, void *, struct pt_regs *);
    unsigned long flags;
    unsigned long mask;
    const char *name;
    void *dev_id;
    struct irqaction *next;
};
```

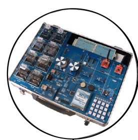
该 irqaction 数据结构中包含对应于此中断处理的相关信息，如中断处理例程的地址，此中断所属的模块名称，以及是否允许共享的标志位，如果允许共享，next 成员将指向共享此中断号的下一个 irqaction 的结构指针等。

另一个重要变量 irq_action 是 irqaction 指针型的，之所以用指针型而不用一个 irqaction 数组，是因为中断个数及它们被如何处理会根据体系结构及系统的变化而变化。Linux 中的中断处理代码是和体系结构相关的，这也意味着 irq_action 数组的大小随于中断源的个数而变化。

中断发生时 Linux 首先读取系统可编程中断控制器中中断状态寄存器，判断出中断源，将其转换成 irq_action 数组中的偏移值。如果此中断没有对应的中断处理过程，则 Linux 核心将记录这个错误，否则，它将调用对应此中断源的所有 irqaction 数据结构中的中断处理例程。

此段功能在 do_irq 函数中完成，代码如下：

华清远见 HQYJ.COM 华清远见教育集团官网：www.hqyj.com




```
    asmlinkage void do_IRQ(int irq, struct pt_regs * regs)
```

其中，第一个参数是中断号，第二个参数是中断发生时 CPU 的现场寄存器状态。根据中断号找到相应的 irqaction 后，执行的关键代码如下：

```
            do {
                status |= action->flags;
                action->handler(irq, action->dev_id, regs);
                action = action->next;
            } while (action);
```

可以看到，内核是凭借 handler 指针调入驱动程序的 ISR，在此使用循环的意义是，如果此中断号被多个 ISR 共享，可以依次遍历每一个注册的 ISR，使它得到一次执行的机会。

当 Linux 核心调用设备驱动的中断处理过程时，此过程必须找出中断产生的原因及相应的解决办法。为了找到设备驱动的中断原因，设备驱动必须读取发生中断的设备上的状态寄存器。设备可能会报告一个错误或通知请求的处理已经完成。如软盘控制器可能报告它已经完成软盘读取磁头对某个扇区的正确定位。一旦确定了中断产生的原因，设备驱动还要完成更多工作，这样 Linux 核心将推迟这些操作。

7.4.2　注册中断处理函数

中断处理程序是管理硬件的驱动程序，每个设备都有与之对应的驱动，驱动注册中断处理函数。驱动可以注册中断处理程序并使能通过这个程序来处理给定的中断线。中断处理函数代码如下：

```
    int request_irq(unsigned int irq, void (*handler)(int irq,void dev_id,struct
pt_regs *regs), unsigned long flags, const char *device, void *dev_id)
    {
        unsigned long retval;
        struct irqaction *action;

        if (irq >= NR_IRQS || !irq_desc[irq].valid || !handler ||
            (irq_flags & SA_SHIRQ && !dev_id))
             return -EINVAL;

        action = (struct irqaction *)kmalloc(sizeof(struct irqaction), GFP_KERNEL);
        if (!action)
             return -ENOMEM;

        action->handler = handler;
        action->flags = irq_flags;
        action->mask = 0;
        action->name = devname;
        action->next = NULL;
        action->dev_id = dev_id;

        retval = setup_arm_irq(irq, action);
        if (retval)
             kfree(action);
        return retval;
    }
```


华清远见 HQYJ.COM　华清远见教育集团官网：www.hqyj.com

参数 irq 表示要申请的硬件中断号。Handler 是向系统登记的中断处理子程序，中断产生时由系统来调用，调用时所带参数 irq 为中断号，dev_id 为申请时告诉系统的设备标识，regs 为中断发生时的寄存器内容。device 为设备名，将会出现在/proc/interrupts 文件中。flag 是申请时的选项，它决定中断处理程序的一些特性，其中，最重要的是中断处理程序是快速处理程序（flag 中设置了 SA_INTERRUPT）还是慢速处理程序（不设置 SA_INTERRUPT）。快速处理程序运行时，所有中断都被屏蔽；而慢速处理程序运行时，除了正在处理的中断外，其他中断都没有被屏蔽。

在 Linux 系统中，中断可以被不同的中断处理程序共享，这要求每一个共享此中断的处理程序申请中断时在 flags 中设置 SA_SHIRQ，这些处理程序之间以 dev_id 来区分。如果中断由某个处理程序独占，则 dev_id 可以为 NULL。从程序中可以看到，进入本函数后首先判断传入的参数是否合法、正确，如果正确，就动态分配一个 irqaction 变量空间，存储到内核中，最后调入硬件平台的相关函数去开启这个中断。如果 request_irq 返回 0 表示成功；返回- INVAL 表示 irq>15 或 handler==NULL；返回-EBUSY 表示中断已经被占用且不能共享。作为系统核心的一部分，设备驱动程序在申请和释放内存时不是调用 malloc 和 free，而是调用 kmalloc 和 kfree。

有注册中断处理程序的函数就一定有取消注册的函数，在 Linux 内核中注销一个中断处理函数，代码如下：

```
void free_irq(unsigned int irq, void *dev_id)
{
    struct irqaction * action, **p;
    unsigned long flags;

    if (irq >= NR_IRQS || !irq_desc[irq].valid) {
        printk(KERN_ERR "Trying to free IRQ%d\n",irq);
#ifdef CONFIG_DEBUG_ERRORS
        __backtrace();
#endif
        return;
    }

    spin_lock_irqsave(&irq_controller_lock, flags);
    for (p = &irq_desc[irq].action; (action = *p) != NULL; p = &action->next) {
        if (action->dev_id != dev_id)
            continue;

        /* Found it - now free it */
        *p = action->next;
        kfree(action);
        goto out;
    }
    printk(KERN_ERR "Trying to free free IRQ%d\n",irq);
#ifdef CONFIG_DEBUG_ERRORS
    __backtrace();
#endif
out:
```


华清远见 HQYJ.COM

华清远见教育集团官网：www.hqyj.com




```
        spin_unlock_irqrestore(&irq_controller_lock, flags);
    }
```

第一个参数为中断号，第二个参数为设备标识，这个参数是为了区别共享同一中断号的不同设备而设的，如果此中断号为一个设备所独享，那么这个参数可为空。

调入函数后，首先判断传入参数是否合法及此中断号的中断是否已经打开，如全部合法，就执行后面的查找符合中断号并符合设备标识的 irqaction 结构体，从链表中摘下后释放其所占的空间。至此，一个中断服务就被彻底释放。

7.4.3 中断标志 flags

中断标志 flags 可以设置为以下 3 项。

- SA_INTERRUPT：如果设置该位，就表示这是一个“快速”中断处理程序；如果清除该位，那么它就是一个“慢速”中断处理程序。
- SA_SHIRQ：该位表明中断可以在设备间共享。共享的概念将在后面的章节中介绍。
- SA_SAMPLE_RANDOM：该位表明产生的中断对/dev/random 和/dev/urandom 设备要使用的熵池（Entropy Pool）有贡献。读这些设备返回真正的随机数，它们用来帮助应用软件选取用于加密的安全钥匙。这些随机数是从一个熵池中取得的，各种随机事件都会对系统的熵池（无序度）有贡献。如果希望设备真正随机地产生中断，应该设置这个标志。而如果中断是可预测的（例如，帧捕捉卡的垂直消隐），那就不值得设置这个标志位，因为它对系统的熵池没有任何贡献。

7.4.4 ISR 上下文

华清远见

当进程发出一个系统调用的请求时，由应用态切换到内核态。这样的内核控制路径称为进程内核路径，又称进程上下文。当 CPU 执行一个与中断有关的内核控制路径时，称为中断上下文。中断的上半部和下半部都属于 ISR 上下文。

7.5 tasklet 机制



tasklet 是一种特殊的软中断机制，它可以被多次调度运行。Linux 内核使用 tasklet 机制实现底半部处理。tasklet 的原意是“小片任务”的意思，这里是指一小段可执行的代码，且通常以函数的形式出现。软中断向量 HI_SOFTIRQ 和 TASKLET_SOFTIRQ 均是用 tasklet 机制来实现的。实际上，现在的底半部处理程序本身就是用 tasklet 实现的。

tasklet 是可以把任务延迟到安全时间执行的一种方式，都在中断期间运行。即使被调度多次，tasklet 也只运行一次，不过 tasklet 可以在 SMP 系统上和其他（不同的）tasklet

华清远见 HQYJ.COM

华清远见教育集团官网：www.hqyj.com

并行运行。在SMP系统上，tasklet还被确保在第一个调度它的 CPU 上运行，因为这样可以提供更好的高速缓存行为，从而提高性能。

每个 tasklet 都与一个函数相联系，当tasklet运行时该函数被调用。该函数只有一个unsigned long类型的参数。把long类型的参数转换为一个指针类型在所有已支持的平台上都是安全的操作。这个tasklet的函数的类型是void，无返回值。

一个tasklet一旦被调度，它就肯定会在一个安全时间运行一次（如果已经被使能）。tasklet可以重新调度自己，其方式和任务队列一样。在多处理器系统上，一个tasklet无须担心自己会在多个处理器上同时运行，因为内核采取了措施确保任何 tasklet都只能在一个地方运行。但是，如果驱动程序中实现了多个tasklet，那么就可能会有多个tasklet在同时运行。在这种情况下，需要使用自旋锁来保护临界区代码（信号量是可以睡眠的，因为tasklet是在中断期间运行的，所以不能用于tasklet）。

从某种程度上讲，tasklet机制是Linux内核对BH机制的一种扩展。在2.4内核引入了softirq机制后，原有的BH机制正是通过tasklet机制这个桥梁来纳入softirq机制的整体框架中的。正是由于这种历史的延伸关系，使得tasklet机制与一般意义上的软中断有所不同，而呈现出以下两个显著的特点：

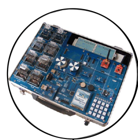
- 与一般的软中断不同，某一段tasklet代码在某个时刻只能在一个CPU上运行，而不像一般的软中断服务函数（即softirq_action结构中的action函数指针）那样，在同一时刻可以被多个CPU并发地执行。
- 与BH机制不同，不同的tasklet代码在同一时刻可以在多个CPU上并发地执行，而不像BH机制那样必须严格地串行化执行（即在同一时刻系统中只能有一个CPU执行BH函数）。

所以，引入tasklet最主要的考虑是为了更好地支持SMP，提高SMP多个CPU的利用率。不同的tasklet可以同时运行于不同的CPU上，但是，同一个tasklet只会在一个CPU上运行。tasklet_struct结构定义在include/linux/interrupt.h文件中，代码如下：

```
struct tasklet_struct
{
      struct tasklet_struct *next; /* 队列指针 */
      unsigned long state;
/* tasklet的状态，按位操作，目前定义了两个位的含义：
TASKLET_STATE_SCHED（第0位）或TASKLET_STATE_RUN（第1位） */
      atomic_t count;              /* 引用计数，通常用1表示disabled */
      void (*func)(unsigned long); /* 函数指针 */
      unsigned long data;          /* func(data) */
};
```

tasklet的使用相当简单。首先，定义一个处理函数void my_tasklet_func(unsigned long)。然后使用DECLARE_TASKLET，将一个tasklet结构与my_tasklet_func(data)函数相关联，代码如下:

```
DECLARE_TASKLET(my_tasklet,my_tasklet_func,data);
/* 定 义 一 个 tasklet 结 构 my_tasklet ， 与 my_tasklet_func(data) 函 数 相 关 联 ， 相 当 于
DECLARE_TASK_QUEUE() */
```




```
    tasklet_schedule(&my_tasklet);
    /* 登记 my_tasklet，允许系统在适当的时候进行调度运行，相当于queue_task(&my_task,&tq_immediate)和mark_bh(IMMEDIATE_BH) */
```

可见 tasklet 的使用很简单，而且 tasklet 还能很好地支持 SMP 结构，因此，在新的 Linux 内核中，tasklet 是建议使用的异步任务执行机制。除了以上提到的使用步骤外，tasklet 机制还提供了另外一些调用接口，如下所示：

```
    DECLARE_TASKLET_DISABLED(name,function,data);
    /*和 DECLARE_TASKLET()类似，不过即使被调度到也不会马上运行，必须等到 enable */
    tasklet_enable(struct tasklet_struct *);
    /* tasklet 使能 */
    tasklet_disble(struct tasklet_struct *);
    /* 禁用 tasklet，只要 tasklet 还没运行，就会推迟到它被 enable */
    tasklet_init(struct tasklet_struct *,void (*func)(unsigned long),unsigned long);
    /* 类似 DECLARE_TASKLET() */
    tasklet_kill(struct tasklet_struct *);
    /* 清除指定 tasklet 的可调度位，即不允许调度该 tasklet，但不做 tasklet 本身的清除 */
```

Linux 系统中定义了两个 tasklet 队列的向量表，每个向量对应一个 CPU（向量表大小为系统能支持的 CPU 最大个数，在 SMP 方式下，目前为 32）组织成一个 tasklet 链表：

```
    struct tasklet_head tasklet_vec[NR_CPUS] __cacheline_aligned;
    struct tasklet_head tasklet_hi_vec[NR_CPUS] __cacheline_aligned;
```

另外，对于 32 个 bottom half，系统也定义了对应的 32 个 tasklet 结构：

```
    struct tasklet_struct bh_task_vec[32];
```

在软中断子系统初始化时，这组 tasklet 的动作被初始化为 bh_action(nr)，而 bh_action(nr)就会调用 bh_base[nr]的函数指针，从而与 bottom half 的语义挂钩。mark_bh(nr)被实现为调用 tasklet_hi_schedule(bh_tasklet_vec+nr)，在这个函数中，bh_tasklet_vec[nr]将被挂接在 tasklet_hi_vec[cpu]链上（其中 CPU 为当前 CPU 编号，也就是说哪个 CPU 提出了 bottom half 的请求，就会在哪个 CPU 上执行该请求），然后激发 HI_SOFTIRQ 软中断信号，从而在 HI_SOFTIRQ 的中断响应中启动运行。

tasklet_schedule(&my_tasklet) 将把 my_tasklet 挂接到 tasklet_vec[cpu]上，激发 TASKLET_SOFTIRQ，在 TASKLET_SOFTIRQ 的中断响应中执行。

注意，一个 tasklet 总是在同一个 CPU 上运行，即使输出来自双 CPU 系统。tasklet 子系统提供了一些其他的函数，用于高级的 tasklet 操作，例如：

```
    void tasklet_disable(struct tasklet_struct *t);
```

这个函数禁止指定的 tasklet。该 tasklet 仍然可以用 tasklet_schedule 调度，但执行被推迟，直到重新被使能。

```
    void tasklet_enable(struct tasklet_struct *t);
```

该函数使能一个先前被禁止的 tasklet。如果该 tastlet 已经被调度，它很快就会运行（但一旦从 tasklet_enable 返回就直接运行了）。

华清远见
HQYJ.COM
华清远见教育集团官网：www.hqyj.com

```
void tasklet_kill(struct tasklet_struct *t);
```

该函数用于对付那些无休止地重新调度自己的 tasklet。tasklet_kill 把指定的 tasklet 从它所在的所有队列中删除。为避免与正在重新调度自己的 tasklet 产生竞态，该函数会等到 tasklet 执行，然后再把它移出队列。这样就可以确保 tasklet 不会在中途被打断。然而，如果目标 tasklet 当前既没有运行也没有重新调度自己，tasklet_kill 将会挂起。tasklet_kill 不能在中断期间被调用。

7.6 上半部和下半部



7.6.1 上半部和下半部的设计

Linux 中的中断处理程序很有特色，它的一个中断处理程序分为两个部分：上半部（top half）和下半部（bottom half）。之所以分为上半部和下半部，完全是考虑到中断处理的效率。

上半部的功能是“登记中断”，当一个中断发生时，它就把设备驱动程序中中断例程的下半部挂到该设备的下半部执行队列中去，然后就等待新的中断的到来。这样一来，上半部执行的速度就会很快，它就可以接受更多设备产生的中断了。上半部之所以要快，是因为它是完全屏蔽中断的，如果它不执行完，其他的中断就不能被及时处理，只能等到这个中断处理程序执行完毕以后。所以，要尽可能多地对设备产生的中断进行服务和处理，中断处理程序就一定要快。

但是，有些中断事件的处理是比较复杂的，所以中断处理程序必须多花一点时间才能够把事情做完。怎么样才能化解在短时间内完成复杂处理的矛盾呢？这时 Linux 引入了下半部的概念。下半部和上半部最大的不同就是，下半部是可中断的，而上半部是不可中断的。下半部几乎做了中断处理程序所有的事情，因为上半部只是将下半部排到了它们所负责的设备的中断处理队列中去，然后就什么都不管了。下半部一般负责的工作是查看设备以获得产生中断的事件信息，并根据这些信息（一般通过读设备上的寄存器得来）进行相应的处理。

由于下半部是可中断的，所以在它运行期间，如果其他设备产生了中断，这个下半部可以暂时中断掉，等到其他设备的上半部运行完了，再回头来运行它。但是有一点一定要注意，那就是如果一个设备中断处理程序正在运行，无论它是运行上半部还是运行下半部，只要中断处理程序还没有处理完毕，在这期间设备产生的新的中断都将被忽略掉。因为中断处理程序是不可重入的，同一个中断处理程序是不能并行的。

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

在处理中断时，中断控制器会屏蔽掉原来发送中断的设备，直到它发送的上一个中断被处理完为止。因此，如果发送中断的那个设备在中断处理期间又发送了一个中断，那么这个中断就永远丢失了。

之所以会发生这种事情，是因为中断控制器并不能缓冲中断信息，所以当前一个中断没有处理完之前又有新的中断到达时，它肯定会丢掉新的中断。但是这种缺陷可以通过设置主处理器（CPU）上的“置中断标志位（STI）”来解决，因为主处理器具有缓冲中断的功能。如果使用了“置中断标志位”，那么在处理完中断以后使用 sti 函数就可以使先前被屏蔽的中断得到服务。

7.6.2　中断处理程序的不可重入性

上面提到有时需要屏蔽中断，这并不是因为技术上实现不了同一中断例程的并行，而是出于管理上的考虑。之所以在中断处理的过程中要屏蔽同一 IRQ 来的新中断，是因为中断处理程序是不可重入的，所以不能并行执行同一个中断处理程序。当驱动程序锁死时，操作系统并不一定会崩溃，但是锁死的驱动程序所支持的那个设备不能再使用了，因为设备驱动程序死了，设备也就死了。

由于设备驱动程序要和设备的寄存器打交道，所以很难写出可以重入的代码，因为设备寄存器就是全局变量。因此，最简捷的办法就是禁止同一设备的中断处理程序并行，即设备的中断处理程序是不可重入的。

另外，必须避免竞争条件的出现，因为一旦竞争条件出现，就有可能会发生死锁的情况，严重时可能会将整个系统锁死。但是，绝大多数由于中断产生的竞争条件，都是在带有中断的内核进程被睡眠造成的。所以，在实现中断时，一定要小心地让进程睡眠，必要时可以使用 cli、sti 或 save_flag、restore_flag。如果数据只是在中断服务例程内部访问，那么就不需要锁，因为内核本身已经保证了中断服务例程不会同时在多个 CPU 上运行。

在 Linux 内核中，bottom half 通常用“BH”表示，最初用于在特权级较低的上下文中完成中断服务的非关键耗时动作，现在也用于一切可在低优先级的上下文中执行的异步动作。最早的 bottom half 实现是借用中断向量表的方式，在目前的 2.x 内核中仍然可以看到，如下所示：

```
static void (*bh_base[32])(void);/* kernel/softirq.c */
```

系统定义了一个函数指针数组，共有 32 个函数指针，采用数组索引来访问，与此相对应的是一套函数：

- void init_bh(int nr,void (*routine)(void));

为第 nr 个函数指针赋值为 routine。

- void remove_bh(int nr);

动作与 init_bh()相反，卸下 nr 函数指针。

- void mark_bh(int nr);

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

标志第 nr 个 bottom half 可执行。由于历史的原因，bh_base 各个函数指针位置大多有了预定义的意义，在 2.4.2 内核中有这样一个枚举，代码如下：

```
enum {
        TIMER_BH = 0,
        TQUEUE_BH,
        DIGI_BH,
        SERIAL_BH,
        RISCOM8_BH,
        SPECIALIX_BH,
        AURORA_BH,
        ESP_BH,
        SCSI_BH,
        IMMEDIATE_BH,
        CYCLADES_BH,
        CM206_BH,
        JS_BH,
        MACSERIAL_BH,
        ISICOM_BH
};
```

约定某个驱动使用某个 bottom half 位置，例如，串口中断就约定使用 SERIAL_BH，现在用得多的主要是 TIMER_BH、TQUEUE_BH 和 IMMEDIATE_BH，但语义已经很不一样了，因为整个 bottom half 的使用方式已经很不一样了，这 3 个函数仅仅是在接口上保持了向下兼容，在实现上一直都在随着内核的软中断机制在变。现在，在 2.6.x 内核中用的是 tasklet 机制。

7.7 工作队列



工作队列（Work Queues）是另一种中断后处理机制。tasklet 虽然也是中断后处理，但是它始终处于中断上下文中，所有的代码必须是原子的。而工作队列函数则处于一个特殊的内核线程上下文中，因此它可以阻塞，代码如下：

```
struct workqueue_struct *create_workqueue(const char *name);
```

这里 name 是工作队列的名字。

工作队列任务可以在编译或运行时创建。任务需要封装在一个 work_struct 结构中。在编译过程中，初始化一个工作队列任务时要用到如下代码：

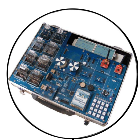
```
DECLARE_WORK(name, void (*function)(void *), void *data);
```

这里，name 是 work_struct 的名字；function 是当任务被调度时调用的函数；data 是指向函数参数的指针。

在运行过程中，初始化一个工作队列时要用到如下代码：

```
INIT_WORK(struct work_struct *work, void (*function)(void *), void *data);
```


华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com



用下面的函数调用来把一个任务加入到工作队列中：

```
    int queue_work(struct workqueue_struct *queue, struct work_struct *work);
    int queue_delayed_work(struct workqueue_struct *queue, struct work_struct *work,
unsigned long delay);
```

在 queue_delayed_work 中指定 delay 是为了保证至少在经过一段给定的最小延时时间以后，工作队列中的任务才可以真正执行。工作队列中的任务由相关的工作线程执行，可能是在一个无法预期的时间，或者是在一段延时以后。任何一个在工作队列中等待的任务可以用下面的方法取消：

```
    int cancel_delayed_work(struct work_struct *work);
```

如果一个取消操作的调用返回时，任务正在执行，那么这个任务将一直执行下去，但不会再加入到队列中。清空工作队列中的所有任务使用如下代码：

```
    void flush_workqueue(struct workqueue_struct *queue);
```

销毁工作队列使用如下代码：

```
    void destroy_workqueue(struct workqueue_struct *queue);
```

不是所有的驱动程序都需要有自己的工作队列。驱动程序可以使用内核提供的默认工作队列。由于这个工作队列由很多驱动程序共享，任务可能需要比较长的一段时间才能执行。为了解决这个问题，可以使工作函数中的延迟保持最小或干脆不要。需要特别注意的是，默认队列对所有的驱动程序来说都是可用的，代码如下：

```
    int schedule_work(struct work_struct *work);     //向工作队列中添加一个任务
```

当模块被加载时，应该调用 flash_scheduled_work 函数，这个函数使等待队列中所有的任务都被执行。

```
    int schedule_delayed_work(struct work_struct *work, unsigned long delay);
    //向工作队列中添加一个任务并延迟执行
```

7.8 系统调用



在操作系统范畴中，我们经常会使用系统调用这一术语。所谓系统调用，就是内核提供的、功能十分强大的一系列函数。这些系统调用是在内核中实现的，再通过一定的方式把系统调用给用户，一般都通过门（Gate）陷入（Trap）实现。系统调用是用户程序和内核交互的接口。系统调用在 Linux 系统中发挥着巨大的作用，如果没有系统调用，那么应用程序就失去了内核的支持。我们在编程时用到的很多函数（如 read、close 等）最终都是在系统调用中实现的。

华清远见 HQYJ.COM 华清远见教育集团官网：www.hqyj.com

7.8.1 初始化系统调用

Linux 系统在 CPU 的保护模式下提供了 4 个特权级别，目前内核都只用到了其中的两个特权级别，分别为“特权级 0”和“特权级 3”，级别 0 就是通常所讲的内核模式，级别 3 就是通常所讲的用户模式。划分这两个级别主要是对系统提供保护。内核模式可以执行一些特权指令和进入用户模式，而用户模式则不能。

每个进程都有自己的地址空间（又称进程空间），进程的地址空间也分为两部分：用户空间和系统空间，在用户模式下只能访问进程的用户空间，在内核模式下则可以访问进程的全部地址空间，这个地址空间中的地址是一个逻辑地址，通过系统段面式的管理机制，访问的实际内存要进行二级地址转换，即：逻辑地址/线性地址/物理地址。

系统调用对于内核来说就相当于函数，关键问题是从用户模式到内核模式的转换、堆栈的切换及参数的传递。

内核在初始化期间会调用 arch/x86/kernel/traps.c 文件中的 trap_init()函数建立 IDT 表，该函数又会调用 set_system_gate(SYSCALL_VECTOR, &system_call)函数来建立 0x80 中断的对应表项。set_system_gate 函数本质上是调用_set_gate(n,DESCTYPE_TRAP| DESCTYPE_DPL3, addr, __KERNEL_CS)函数，将 0x80 中断的相应属性进行如下初始化：

- 段选择子为内核代码段__KERNEL_CS。
- 段偏移量指向 arch/x86/kernel/entry.S 文件中的 system_call 汇编代码段。
- 将该项中断的属性设置成 DECTYPE_TRAP（表示该中断为陷阱，相应处理程序中不会禁止可屏蔽中断）和 DESCTYPE_DPL3（允许用户态进程调用该中断）。

trap_init 函数原型如下：

```
void __init trap_init(void)
{
#ifdef CONFIG_X86_32
    int i;
#endif

#ifdef CONFIG_EISA
    void __iomem *p = early_ioremap(0x0FFFD9, 4);

    if (readl(p) == 'E' + ('I'<<8) + ('S'<<16) + ('A'<<24))
        EISA_bus = 1;
    early_iounmap(p, 4);
#endif

    set_intr_gate(0, &divide_error);
    set_intr_gate_ist(1, &debug, DEBUG_STACK);
    set_intr_gate_ist(2, &nmi, NMI_STACK);
    /* int3 can be called from all */
    set_system_intr_gate_ist(3, &int3, DEBUG_STACK);
```


华清远见 HQYJ.COM 华清远见教育集团官网：www.hqyj.com




```
	/* int4 can be called from all */
	set_system_intr_gate(4, &overflow);
	set_intr_gate(5, &bounds);
	set_intr_gate(6, &invalid_op);
	set_intr_gate(7, &device_not_available);
#ifdef CONFIG_X86_32
	set_task_gate(8, GDT_ENTRY_DOUBLEFAULT_TSS);
#else
	set_intr_gate_ist(8, &double_fault, DOUBLEFAULT_STACK);
#endif
	set_intr_gate(9, &coprocessor_segment_overrun);
	set_intr_gate(10, &invalid_TSS);
	set_intr_gate(11, &segment_not_present);
	set_intr_gate_ist(12, &stack_segment, STACKFAULT_STACK);
	set_intr_gate(13, &general_protection);
	set_intr_gate(14, &page_fault);
	set_intr_gate(15, &spurious_interrupt_bug);
	set_intr_gate(16, &coprocessor_error);
	set_intr_gate(17, &alignment_check);
#ifdef CONFIG_X86_MCE
	set_intr_gate_ist(18, &machine_check, MCE_STACK);
#endif
	set_intr_gate(19, &simd_coprocessor_error);

#ifdef CONFIG_IA32_EMULATION
	set_system_intr_gate(IA32_SYSCALL_VECTOR, ia32_syscall);
#endif

#ifdef CONFIG_X86_32
	if (cpu_has_fxsr) {
		printk(KERN_INFO "Enabling fast FPU save and restore... ");
		set_in_cr4(X86_CR4_OSFXSR);
		printk("done.\n");
	}
	if (cpu_has_xmm) {
		printk(KERN_INFO
			"Enabling unmasked SIMD FPU exception support... ");
		set_in_cr4(X86_CR4_OSXMMEXCPT);
		printk("done.\n");
	}

	set_system_trap_gate(SYSCALL_VECTOR, &system_call);

	/* Reserve all the builtin and the syscall vector: */
	for (i = 0; i < FIRST_EXTERNAL_VECTOR; i++)
		set_bit(i, used_vectors);

	set_bit(SYSCALL_VECTOR, used_vectors);
#endif
	/*
	 * Should be a barrier for any external CPU state:
	 */
	cpu_init();
```

```
#ifdef CONFIG_X86_32
    trap_init_hook();
#endif
}
```

7.8.2 system_call 函数

这个函数位于 arch/x86/kernel/entry.S 文件中，system_call 汇编代码段运行于系统核心态，对系统调用进行预处理，并调用相应的服务例程进行处理，其主要工作如下：

- 使用 SAVA_ALL 宏将要用到的所有寄存器保存到栈中（不包括已由硬件自动保存的寄存器）。
- 通过获得内核栈指针并对 8KB 取整在 ebp 中放入进程描述符地址。
- 检查当前进程的进程描述符的 ptrace 字段的 PF_TRACESYS 标志，若为 1，则在运行服务例程前调用一次 syscall_trace_entry 来允许调试程序吸收进程信息。
- 检查系统调用号的有效性，若不小于 nr_syscalls，则报错并返回-ENOSYS，否则通过在 sys_call_table 中查找服务例程的地址来调用特定的服务例程。
- 将返回值存放在 eax 中，并通过跳转到 resume_userspace 来返回到用户态。

相关代码如下：

```
ENTRY(system_call)
    RING0_INT_FRAME            # can't unwind into user space anyway
    pushl %eax                 # save orig_eax
    CFI_ADJUST_CFA_OFFSET 4
    SAVE_ALL
    GET_THREAD_INFO(%ebp)
                       # system call tracing in operation / emulation
    testw $_TIF_WORK_SYSCALL_ENTRY,TI_flags(%ebp)
    jnz syscall_trace_entry
    cmpl $(nr_syscalls), %eax
    jae syscall_badsys
syscall_call:
    call *sys_call_table(,%eax,4)
    movl %eax,PT_EAX(%esp)          # store the return value
syscall_exit:
    LOCKDEP_SYS_EXIT
    DISABLE_INTERRUPTS(CLBR_ANY)    # make sure we don't miss an interrupt
                       # setting need_resched or sigpending
                       # between sampling and the iret
    TRACE_IRQS_OFF
    movl TI_flags(%ebp), %ecx
    testw $_TIF_ALLWORK_MASK, %cx  # current->work
    jne syscall_exit_work

restore_all:
    movl PT_EFLAGS(%esp), %eax# mix EFLAGS, SS and CS
    # Warning: PT_OLDSS(%esp) contains the wrong/random values if we
    # are returning to the kernel.
```


华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com




```
    # See comments in process.c:copy_thread() for details.
    movb PT_OLDSS(%esp), %ah
    movb PT_CS(%esp), %al
    andl $(X86_EFLAGS_VM | (SEGMENT_TI_MASK << 8) | SEGMENT_RPL_MASK), %eax
    cmpl $((SEGMENT_LDT << 8) | USER_RPL), %eax
    CFI_REMEMBER_STATE
    je ldt_ss            # returning to user-space with LDT SS
restore_nocheck:
    TRACE_IRQS_IRET
restore_nocheck_notrace:
    RESTORE_REGS
    addl $4, %esp        # skip orig_eax/error_code
    CFI_ADJUST_CFA_OFFSET -4
irq_return:
    INTERRUPT_RETURN
.section .fixup,"ax"
ENTRY(iret_exc)
    pushl $0         # no error code
    pushl $do_iret_error
    jmp error_code
.previous
.section __ex_table,"a"
    .align 4
    .long irq_return,iret_exc
.previous

    CFI_RESTORE_STATE
ldt_ss:
    larl PT_OLDSS(%esp), %eax
    jnz restore_nocheck
    testl $0x00400000, %eax      # returning to 32bit stack?
    jnz restore_nocheck          # allright, normal return

#ifdef CONFIG_PARAVIRT
    /*
     * The kernel can't run on a non-flat stack if paravirt mode
     * is active.  Rather than try to fixup the high bits of
     * ESP, bypass this code entirely.  This may break DOSemu
     * and/or Wine support in a paravirt VM, although the option
     * is still available to implement the setting of the high
     * 16-bits in the INTERRUPT_RETURN paravirt-op.
     */
    cmpl $0, pv_info+PARAVIRT_enabled
    jne restore_nocheck
#endif

    /* If returning to userspace with 16bit stack,
     * try to fix the higher word of ESP, as the CPU
     * won't restore it.
     * This is an "official" bug of all the x86-compatible
     * CPUs, which we can try to work around to make
     * dosemu and wine happy. */
    movl PT_OLDESP(%esp), %eax
```


华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

```
    movl %esp, %edx
    call patch_espfix_desc
    pushl $__ESPFIX_SS
    CFI_ADJUST_CFA_OFFSET 4
    pushl %eax
    CFI_ADJUST_CFA_OFFSET 4
    DISABLE_INTERRUPTS(CLBR_EAX)
    TRACE_IRQS_OFF
    lss (%esp), %esp
    CFI_ADJUST_CFA_OFFSET -8
    jmp restore_nocheck
    CFI_ENDPROC
ENDPROC(system_call)

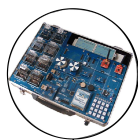
work_pending:
    testb $_TIF_NEED_RESCHED, %cl
    jz work_notifysig
work_resched:
    call schedule
    LOCKDEP_SYS_EXIT
    DISABLE_INTERRUPTS(CLBR_ANY)   # make sure we don't miss an interrupt
                                   # setting need_resched or sigpending
                                   # between sampling and the iret
    TRACE_IRQS_OFF
    movl TI_flags(%ebp), %ecx
    andl $_TIF_WORK_MASK, %ecx# is there any work to be done other
                                   # than syscall tracing?
    jz restore_all
    testb $_TIF_NEED_RESCHED, %cl
    jnz work_resched

work_notifysig:                    # deal with pending signals and
                                   # notify-resume requests
#ifdef CONFIG_VM86
    testl $X86_EFLAGS_VM, PT_EFLAGS(%esp)
    movl %esp, %eax
    jne work_notifysig_v86         # returning to kernel-space or
                                   # vm86-space
    xorl %edx, %edx
    call do_notify_resume
    jmp resume_userspace_sig

    ALIGN
work_notifysig_v86:
    pushl %ecx                     # save ti_flags for do_notify_resume
    CFI_ADJUST_CFA_OFFSET 4
    call save_v86_state            # %eax contains pt_regs pointer
    popl %ecx
    CFI_ADJUST_CFA_OFFSET -4
    movl %eax, %esp
#else
    movl %esp, %eax
#endif
```


华清远见 HQYJ.COM

华清远见教育集团官网：www.hqyj.com




```
    xorl %edx, %edx
    call do_notify_resume
    jmp resume_userspace_sig
END(work_pending)
```

7.8.3 参数的传递与验证

Linux 的系统调用的参数是在调用 0x80 号中断前通过寄存器进行传递的，这一工作由库函数完成。传递的参数最多只能有 6 个，依次存入寄存器 eax（系统调用号）、ebx、ecx、edx、esi 和 edi 中。对于长度大于 32 位的参数，则传递其地址；若需要传递的参数多于 6 个，则传递的参数为参数所在的内存区。系统的返回值存放于 eax 中返回。系统调用号在 include/asm-i386/unistd.h 文件中定义。

执行实际的系统调用服务例程前，Linux 内核会先对传递给系统调用的所有参数进行检测，包括参数的类型、权限限制等，如果出错则返回-EBADF。目前，内核主要执行地址检查，即检查要访问的地址空间是否属于该进程，并且该地址不得在内核区间内。地址检查通过 include/asm-i386/uaccess.h 文件中的 access_ok 和__addr_ok 两个宏实现。

```
#define __addr_ok(addr) (!((unsigned long)(addr) & (current->addr_limit.seg)))

#define __range_ok(addr,size) ({ \
    unsigned long flag,sum; \
    asm("addl %3,%1 ; sbbl %0,%0; cmpl %1,%4; sbbl $0,%0" \
        :"=&r" (flag), "=r" (sum) \
        :"1" (addr),"g" ((int)(size)),"g" (current->addr_limit.seg)); \
    flag; })
#ifdef CONFIG_X86_WP_WORKS_OK
#define access_ok(type,addr,size) (__range_ok(addr,size) == 0)
#else
#define access_ok(type,addr,size) ( (__range_ok(addr,size) == 0) && \
             ((type) == VERIFY_READ || boot_cpu_data.wp_works_ok || \
             segment_eq(get_fs(),KERNEL_DS) || \
              __verify_write((void *)(addr),(size))))
#endif
```

打开编译目录的 autoconf.h，其中有如下定义：

```
#define CONFIG_X86_WP_WORKS_OK 1
    if (!access_ok (VERIFY_WRITE, to, size))
        return -EFAULT;
```

因此编绎成：

```
    movl $-8192,% eax
    andl % esp,% eax      ;取当前 task_struct
    movl to,% ebx
    addl size,% ebx
    sbbl % edx,% edx      ;如果 addl 溢出,edx 为-1
    cmpl % ebx,12(% eax)  ;current->addr_limit.seg - ebx
    sbbl $0,% edx     ;如果 cmpl 为负,edx 减 1
    testl % edx,% edx
```


华清远见 HQYJ.COM 华清远见教育集团官网：www.hqyj.com

```
        je 1f                    ;edx 等于 0 正常
        movl $-14,% eax              ;故障返回
    1:
        正常运行
        ...
```

由此可见，access_ok 检测了(to+size)溢出和((to+size)>addr_limit.seg)这两种非法情况。

7.9 本章习题

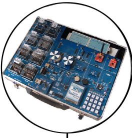


1. 什么是中断？
2. 什么是中断处理程序？编写中断处理程序时需要注意哪些问题？
3. 什么是操作系统的系统调用？
4. 如何为 Linux 系统增加一个系统调用

华清远见
HQYJ.COM
华清远见

第 8 章 文件管理

文件是计算机系统的软件资源，操作系统本身和大量的用户程序、数据都是以文件形式组织和存放的，对这些资源的有效管理和充分利用是操作系统的重要任务之一。本章介绍文件和文件系统的概念，并对操作系统的文件管理功能进行简要说明。



华清远见

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

8.1 磁盘的物理组织



文件系统主要完成对磁盘上的数据和程序进行管理，所以首先需要了解磁盘（硬盘）的结构。

磁盘又称硬盘，由一组盘片组成。每个盘片的上下两面都涂有磁粉，磁化后可以存储信息数据。每个盘片的上下两面都安装有磁头。磁头被安装在梳状的可以做直线运动的小车上以便寻道。每个盘面被格式化成有若干条同心圆的磁道，并规定最外面的磁道是 0 磁道，次外层是 1 磁道。每个磁道又被分成若干个扇区，并顺序排为 1 号扇区、2 号扇区，等等。通常，一个扇区可以存储 512B 的二进制信息位。这也是 CPU 进行磁盘 I/O 操作时能够读出和写入的最小单位。每个盘面上的同号磁道组成一个柱面，也就是说每个盘面的 0 号磁道组成 0 号柱面，所有的 1 号磁道组成 1 号柱面，等等。硬盘结构示意图如图 8.1 所示。

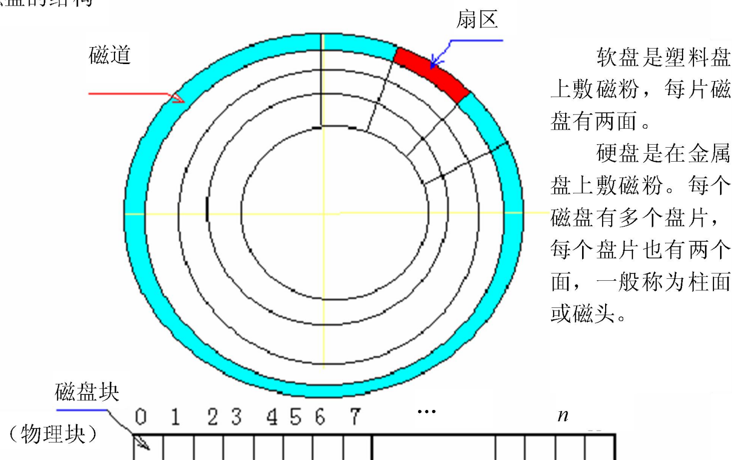
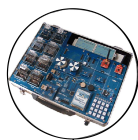


图 8.1　硬盘结构示意图

在 Linux 操作系统中，内核采用的方法是把物理磁盘抽象为逻辑磁盘来管理文件系统。所谓逻辑磁盘，是把物理磁盘按照磁头号、磁道号、扇区号及盘面号划分成磁盘块的线性数组（也叫线性序列）。例如，把 1 号盘面 0 号磁道的 0 号扇区定义为 0 号磁盘块；把 1 号盘面 0 号磁道的 1 号扇区定义为 1 号磁盘块，等等；当然，一个磁盘块可以包含多个扇区，一般扇区数是 2 的整次幂。显然，当把实际的磁盘看成是磁盘块的线性数组时，就把物理磁盘存储数据的实际地址（即磁道号、扇区号及盘面号）隐藏起来。

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com



因此，呈现在系统高层面前的已经不是物理磁盘，而是一个经过加工以后的逻辑磁盘。图 8.2 给出了逻辑磁盘的结构示意图，它比物理硬盘的结构要简单得多。当系统执行磁盘 I/O 操作时，系统给出试图访问的逻辑磁盘块号。由设备驱动程序根据该块号计算出物理磁盘的磁道号、磁头号及扇区号，然后启动硬盘把磁头向前或向后移动到相应的柱面，这就是所谓的寻道。寻道是磁盘 I/O 操作中最为耗时的一个操作。一旦磁头找到磁道，并且相应的扇区转到磁头下面，数据传输就开始了。

8.2 文件和目录



8.2.1 文件的分类

文件是一个具有符号的一组相关联元素的有序序列。文件可以包含范围非常广泛的内容。系统和用户都可以将具有一定独立功能的程序模块、一组数据或一组文字命名为一个文件。文件是以单个名称在计算机上存储的信息集合。文件可以是文本文档、图片、程序，等等。文件通常具有 3 个字母的文件扩展名，用于指示文件类型。

文件有很多种，运行的方式也各有不同。一般来说，我们可以通过文件名来识别这个文件是哪种类型，特定的文件都会有特定的图标，也只有安装了相应的软件，才能正确显示这个文件的图标。

在计算机系统中存放着大量的文件，它们有着不同的内容、用途和形式。

为了便于对文件进行整理和加工，通常把众多的文件从不同的角度进行分类。下面介绍几种经常使用的文件分类方法。

1. 按文件的创建角度分类

按文件的创建角度，文件可以分为以下 3 类。

- **系统文件**：即由操作系统创建的文件。这些文件包含操作系统执行的程序和处理的数据。系统文件仅供操作系统使用，不对用户开放。
- **用户文件**：即由用户创建的文件。这些文件包含的是用户的信息，如用户的程序、数据和其他各种形式的信息。
- **库文件**：即由系统创建、供系统和用户使用的文件。它们是一些由标准函数或子程序及常用的应用程序组成的文件。库文件允许用户调用，但是不允许用户修改。在某些系统中，允许用户通过系统向库文件中添加信息。

2. 按文件的读取权限分类

按文件的读取权限，文件可以分为以下 4 类。

- **只读文件**：指只允许对文件进行读操作，而不允许写操作的文件。

华清远见

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

- **读写文件**：指既可以进行读操作，又可以进行写操作的文件。
- **可执行文件**：指只可以调入到内存中执行，而不能对它们进行读/写操作的文件。
- **不保护文件**：这种文件不进行任何保护，所有用户都可以使用。

3．按文件在系统中的信息流向分类

按文件在系统中的信息流向，文件可以分为以下 3 类。

- **输入文件**：这种文件只能从输入设备中读入到内存，如读卡器上的文件。
- **输出文件**：这种文件只能从系统写入到输出设备中，如打印机上的文件。
- **输入/输出文件**：这种文件既可以从输入设备中读取，又可以向输出设备写入，如磁盘上的文件。

4．按文件信息的逻辑结构分类

按文件信息的逻辑结构，文件可以分为以下两类。

- **流文件**：以字符为基本单位，按照一定顺序组成的文件。文件内的信息就是一连串的字符，不再划分结构。它使用结束符来标识文件的结束。
- **记录文件**：把文件内的信息划分成多个记录，记录是文件组织和操作的基本单位。记录文件可以分为下面两种。
- **文本文件**：即由字符代码组成的文件。文本文件可以直接显示在屏幕上，或者在打印机上打印，也可以使用编辑器进行编辑。
- **二进制文件**：即由二进制数据组成的文件，如可执行程序、图像文件、声音文件等。二进制文件不能直接显示或打印。

8.2.2 目录

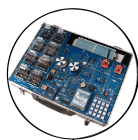
华清远见

目录是一类特殊的文件，利用它可以构成文件系统的分层树形结构。像同普通文件那样，目录文件也包含数据；目录文件与普通文件的差别是，核心对这些数据加以结构化，它是由成对的“i 节点号/文件名”构成的列表。

- i 节点号是检索 I 节点表的下标，i 节点中存放有文件的状态信息。
- 文件名是给一个文件分配的文本形式的字符串，用来标识该文件。在一个指定的目录中，任何两项都不能重名。

每个目录的第一项都表示目录本身，并以“.”作为它的文件名。每个目录的第二项的名字是“..”，表示该目录的父目录。当把文件添加到一个目录中时，该目录的大小会增长，以便容纳新文件名。当删除文件时，目录的尺寸并不减少，而是核心对该目录项做上特殊标记，以便下次添加一个文件时重新使用它。

Linux 文件系统采用带链接的树形目录结构，即只有一个根目录（通常用“/”表示），其中含有下级子目录或文件的信息；子目录中又可含有更下级的子目录或文件的信息。这样一层一层地延伸下去，构成一棵倒置的树，如图 8.3 所示。



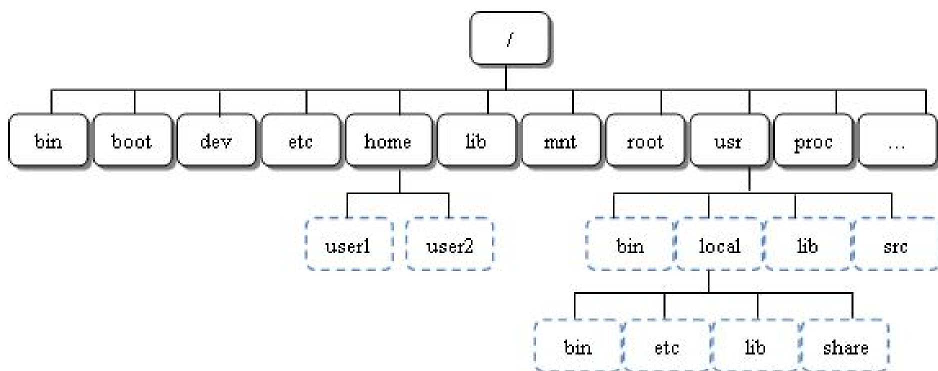


图 8.3　Linux 系统目录

在目录树中，根节点和中间节点都必须是目录，而普通文件和特别文件只能作为“叶子”出现。当然，目录也可以作为叶子。

8.2.3　文件系统

文件系统是指文件存在的物理空间。在 Linux 系统中，每个分区都是一个文件系统，都有自己的目录层次结构。Linux 重要的特征之一就是支持多种文件系统。虽然不同文件系统的实现方法和访问方式各不相同，但是 Linux 在底层文件系统的基础上进行了封装，为内核其他模块和应用程序提供了一致的访问接口。

大部分 Linux 文件系统种类具有类似的通用结构，即使细节有些变化。其中心概念是超级块（Super Block）、i 节点（Inode）、数据块（Data Block）、目录块（Directory Block）和间接块（Indirection Block）。超级块包括文件系统的总体信息，如大小（其准确信息依赖文件系统）。i 节点包括除了名字外的一个文件的所有信息，名字与 i 节点数目一起存在目录中，目录条目包括文件名和文件的 i 节点数目。i 节点包括几个数据块的数目，用于存储文件的数据。i 节点中只有少量数据块数的空间，如果需要更多，会动态分配指向数据块的指针空间。这些动态分配的块是间接块；为了找到数据块，i 节点包括除了名字指出它必须先找到间接块的号码。

Linux 支持多种操作系统，其实现机制在后面的章节将会讲到。

8.3　虚拟文件系统



Linux 系统允许众多不同种类的文件系统共存，如 Ext3、VFAT 等。通过使用同一套文件 I/O 系统调用，即可对 Linux 中的任意文件进行操作，而无须考虑其所在的文件

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

系统的具体格式。此外，Linux 还支持跨文件系统的文件操作，即对文件的操作可以跨文件系统进行，如图 8.3 所示。实现这种操作的机制正是虚拟文件系统。



图 8.4　跨文件系统操作

8.3.1　虚拟文件系统概述

Linux 文件系统分为 3 个部分，第一部分是 Virtual File System（VFS），它是 Linux 文件系统对外的接口，任何要使用文件系统的程序都必须经由这层接口来使用它。另外两部分是属于文件系统的内部实现，分别是 Cache 和真正的文件系统（如 Ext3、VFAT 等）。

虚拟文件系统是 Linux 内核中的一个软件抽象层，它一方面用于给用户空间的程序提供文件系统接口，另一方面还提供了内核中的一个抽象功能，它通过一些数据结构及其方法向实际的文件系统提供接口，实现不同文件系统在 Linux 中共存。系统中所有文件系统不但依赖于 VFS 共存，同时也要依靠 VFS 协同工作。

为了能支持各种文件系统，VFS 定义了所有文件系统都必须支持基本的、概念上的接口和数据结构，例如，超级块、节点、文件操作函数入口等。换句话说，一个实际的文件系统要想被 Linux 支持，就必须提供一个符合 VFS 标准的接口，这样才能与 VFS 协同工作。VFS 不是实际的操作系统，它只是一种转换机制，仅存在于内存中，不存在于任何外存空间。如图 8.5 所示为 VFS 在内核中与实际的文件系统的协同关系。

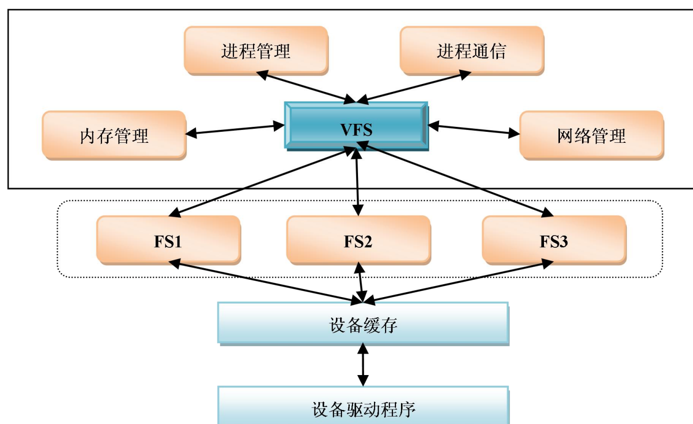


图 8.5　VFS 在内核中与实际文件系统的协同关系图

VFS 主要提供以下一些功能：

- 记录可用的文件系统类型。
- 将设备与对应的文件系统相联系。
- 处理一些面向文件的通用操作。
- 涉及针对文件系统的操作时，VFS 把它们映射到与控制文件、目录及 inode 相关的物理文件系统。

8.3.2 VFS 超级块

VFS 主要通过 4 个主要的数据结构及一些辅助的数据结构描述其结构信息。其中最重要的是 VFS 超级块。VFS 超级块用于存储一个已安装的文件系统的控制信息，代表一个已经安装的文件系统，包含以下主要信息。

- 设备标识符。这是存储文件系统的物理块设备的设备标识符，如系统中第一个 IDE 磁盘/dev/hda1 的标识符是 0x301。
- 索引节点指针。安装索引节点指针指向被安装的子文件系统的第一个索引节点；覆盖索引节点指针指向安装文件系统目录（安装点）的索引节点。根文件系统的 VFS 超级块中没有覆盖索引节点指针。
- 数据块大小。文件系统中数据块的字节数。
- 超级块操作集。指向一组超级块操作例程的指针，VFS 利用它们可以读/写索引节点和超级块。
- 文件系统类型。指向所安装的文件系统类型的指针。
- 文件系统的特殊信息。指向文件系统所需要信息的指针。

超级块数据结构在 include\linux\fs.h 中定义，关于结构中各成员描述如表 8.1 所示。

```
struct super_block {
    struct list_head  s_list;       /* Keep this first */
    dev_t             s_dev;        /* search index; _not_ kdev_t */
    unsigned long     s_blocksize;
    unsigned char     s_blocksize_bits;
    unsigned char     s_dirt;
    unsigned long long    s_maxbytes;  /* Max file size */
    struct file_system_type    *s_type;
    const struct super_operations  *s_op;
    struct dquot_operations    *dq_op;
    struct quotactl_ops    *s_qcop;
    const struct export_operations *s_export_op;
    unsigned long     s_flags;
    unsigned long     s_magic;
    struct dentry     *s_root;
    struct rw_semaphore    s_umount;
    struct mutex      s_lock;
    int           s_count;
    int           s_syncing;
    int           s_need_sync_fs;
```

```
    atomic_t        s_active;
#ifdef CONFIG_SECURITY
    void                    *s_security;
#endif
    struct xattr_handler  **s_xattr;

    struct list_head  s_inodes;    /* all inodes */
    struct list_head  s_dirty; /* dirty inodes */
    struct list_head  s_io;        /* parked for writeback */
    struct list_head  s_more_io;   /* parked for more writeback */
    struct hlist_head s_anon;      /* anonymous dentries for (nfs) exporting */
    struct list_head  s_files;
    /* s_dentry_lru and s_nr_dentry_unused are protected by dcache_lock */
    struct list_head  s_dentry_lru;     /* unused dentry lru */
    int             s_nr_dentry_unused;    /* # of dentry on lru */

    struct block_device   *s_bdev;
    struct mtd_info       *s_mtd;
    struct list_head  s_instances;
    struct quota_info s_dquot; /* Diskquota specific options */

    int             s_frozen;
    wait_queue_head_t s_wait_unfrozen;

    char s_id[32];                 /* Informational name */

    void              *s_fs_info;  /* Filesystem private info */
    fmode_t           s_mode;

    /*
     * The next field is for VFS *only*. No filesystems have any business
     * even looking at it. You had been warned.
     */
    struct mutex s_vfs_rename_mutex;    /* Kludge */

    /* Granularity of c/m/atime in ns.
       Cannot be worse than a second */
    u32          s_time_gran;

    /*
     * Filesystem subtype.  If non-empty the filesystem type field
     * in /proc/mounts will be "type.subtype"
     */
    char *s_subtype;

    /*
     * Saved mount options for lazy filesystems using
     * generic_show_options()
     */
    char *s_options;
};
```


华清远见
HQYJ.COM
华清远见教育集团官网：www.hqyj.com

表 8.1　SUPER_BLOCK 数据结构成员说明

成　　员	描　　述
s_list	所有超级块通过 s_list 链接到 super_block 结构全局链表中
s_dev	存放该超级块的设备 ID
s_blocksize	超级块大小，字节数
s_blocksize_bits	超级块大小，位数
s_dirt	超级块的文件修改标识
s_maxbytes	超级块中文件的最大长度
*s_type	超级块所属的文件系统类型描述结构指针
*s_op	该超级块的操作函数组结构指针
*dq_op	超级块的磁盘配额操作函数组结构指针
*s_qcop	用于超级块的磁盘配额控制的函数组结构指针
*s_export_op	文件系统输出操作函数组结构指针
s_flags	超级块的挂接标识字
s_magic	魔数
*s_root	根目录条目描述结构指针
s_umount	卸载读/写信号量
s_lock	超级块信号量
s_count	引用计数
s_syncing	同步标识
s_need_sync_fs	对已挂接文件系统进行同步的标识
s_active	活动计数器
*s_security	安全相关
**s_xattr	指向超级块扩展属性结构的指针
s_inodes	系统中的所有节点
s_dirty	系统中的所有脏节点
s_io	等待被写入磁盘的索引节点的链表
s_more_io	更多等待被写入磁盘的索引节点的链表
s_anon	匿名项
s_files	文件数
s_dentry_lru	LRU 链表中未用到的目录项
s_nr_dentry_unused	LRU 链表中未用到的目录项数目
*s_bdev	块设备
*s_mtd	MTD 设备
s_instances	该类型文件系统

华清远见

续表

成　　员	描　　述
s_dquot	磁盘配额相关函数
s_frozen	用于冻结文件系统的标识
s_wait_unfrozen	进程挂起的等待队列，直到文件系统解冻
s_id[32];	包含该超级块的设备名称
*s_fs_info	文件系统的私有数据
s_mode	模式相关
s_vfs_rename_mutex	VFS 通过目录重命名文件时使用的信号量
s_time_gran	时间戳
*s_subtype	文件系统的子类型
*s_options	使用 generic_show_options()时，用这个成员保存挂载时的选项

超级块描述了一个文件系统的信息，对文件系统的操作，操作系统内核还提供了超级块操作函数组结构——super_operations。这个函数组中提供了文件系统操作的接口，由开发文件系统的开发人员完成。其结构原型如下（include\linux\fs.h）：

```
struct super_operations {
    struct inode *(*alloc_inode)(struct super_block *sb);
    void (*destroy_inode)(struct inode *);
    void (*dirty_inode) (struct inode *);
    int (*write_inode) (struct inode *, int);
    void (*drop_inode) (struct inode *);
    void (*delete_inode) (struct inode *);
    void (*put_super) (struct super_block *);
    void (*write_super) (struct super_block *);
    int (*sync_fs)(struct super_block *sb, int wait);
    void (*write_super_lockfs) (struct super_block *);
    void (*unlockfs) (struct super_block *);
    int (*statfs) (struct dentry *, struct kstatfs *);
    int (*remount_fs) (struct super_block *, int *, char *);
    void (*clear_inode) (struct inode *);
    void (*umount_begin) (struct super_block *);
    int (*show_options)(struct seq_file *, struct vfsmount *);
    int (*show_stats)(struct seq_file *, struct vfsmount *);
#ifdef CONFIG_QUOTA
    ssize_t (*quota_read)(struct super_block *, int, char *, size_t, loff_t);
    ssize_t (*quota_write)(struct super_block *, int, const char *, size_t,
    loff_t);
#endif
    int (*bdev_try_to_free_page)(struct super_block*, struct page*, gfp_t);
};
```

SUPER_OPERATIONS 数据结构成员说明如表 8.2 所示。

华清远见 HQYJ.COM　华清远见教育集团官网：www.hqyj.com

表 8.2　SUPER_OPERATIONS 数据结构成员说明

成　　员	描　　述
alloc_inode	为索引节点描述结构分配空间时调用
destroy_inode	删除索引节点描述结构时调用
dirty_inode	当索引节点被标记为脏时调用
write_inode	更新索引节点时调用
drop_inode	撤销索引节点时调用，但不真正删除
delete_inode	真正删除索引节点时调用
put_super	释放超级块
write_super	更新文件系统超级块
sync_fs	更新磁盘上的具体文件系统数据结构时调用
write_super_lockfs	阻塞对文件系统的修改，并更新超级块
unlockfs	取消由 write_super_lockfs 方法对文件系统更新的阻塞
Statfs	统计文件系统信息
remount_fs	重新挂载文件系统时调用
clear_inode	撤销磁盘索引节点执行具体文件系统操作时调用
umount_begin	卸载文件系统时调用
show_options	显示文件系统选项时调用
show_stats	显示文件系统信息时调用
quota_read	配额读取时调用
quota_write	配额写入时调用

例如，我们写一个文件系统 myfs，其中的一个工作是设计一个 myfs_sops，myfs_sops 变量的成员值如下：

```
static struct super_operations myfs_sops = {
    read_inode:   myfs_read_inode,
    write_inode:  myfs_write_inode,
    put_inode:    myfs_put_inode,
    delete_inode: myfs_delete_inode,
    put_super:     myfs_put_super,
    write_super:  myfs_write_super,
    ...
  };
```

当第一次挂载这个文件系统分区时会调用 myfs _read_super()，该函数有一条语句：

```
sb->s_op = & myfs _sops。
```

super_operations 就是一个**抽象类**，它只提供接口但并没有实现这些接口，而 myfs_sops 则是**具体类**，实现相应的接口。

华清远见

8.3.3　文件控制块

为了便于对文件进行控制和管理，文件系统为每个文件唯一地设置一个文件控制块（File Control Block，FCB）。文件控制块是文件存在的标志。它是操作系统为管理文件而设置的数据结构，存放了为管理文件所需的所有有关信息。文件控制块的作用就是操作系统和要处理的文件之间相联系的一个纽带，操作系统要依靠 FCB 中的数据完成对文件的读/写操作。

简单地说，一个文件控制块就是一个文件目录项。一个文件目录也被看做是一个文件，称为目录文件。

文件控制块通常包括下面一些信息：

- 文件名。
- 文件类型。
- 物理地址。
- 文件大小。
- 保护信息。
- 使用计数。
- 时间。

文件控制块包含文件除数据以外的控制信息，例如，有关文件存取控制的信息（文件名、存取权限等）、有关文件逻辑结构和物理结构的信息、有关文件管理的信息。因此，一个文件控制块的存储空间是很大的。如果将文件控制块都直接保存在目录文件中，那么目录文件将会占据较多的物理块。遍历目录时就会读取多个物理块，从而降低了检索文件的速度。为了解决这个问题，Linux 引入了名为索引节点的数据结构。

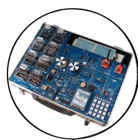
为了减少目录文件所占的物理块，Linux 将文件控制块一分为二，第一部分是文件名，第二部分是其他信息。其他信息被组织成定长的数据结构，称为索引节点。每个索引节点都有一个编号，称为索引号。每个文件目录项只保存文件名及文件名对应的索引号。这样，文件目录项中只剩下 14B 的文件名和 2B 的索引号，一个 512B 的物理块就可以保存 32 个文件目录项。也就是说，一个包含 32 个文件或子目录的目录，只占据 1 个物理块。可见，目录文件占据的物理块被大大减少。如图 8.6 所示为使用了索引节点的物理文件的物理结构示意图。

文件名 1	索引号 1
文件名 2	索引号 2
⋮	⋮
文件名 *n*	索引号 *n*

图 8.6　使用了索引节点的物理文件的物理结构

华清远见

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com



inode 数据结构是 Linux 文件系统中最重要的一个数据结构，它在 include\linux\fs.h 文件中实现，代码如下：

```
struct inode {
    struct hlist_node i_hash;
    struct list_head  i_list;
    struct list_head  i_sb_list;
    struct list_head  i_dentry;
    unsigned long     i_ino;
    atomic_t          i_count;
    unsigned int      i_nlink;
    uid_t             i_uid;
    gid_t             i_gid;
    dev_t             i_rdev;
    u64               i_version;
    loff_t            i_size;
#ifdef __NEED_I_SIZE_ORDERED
    seqcount_t        i_size_seqcount;
#endif
    struct timespec       i_atime;
    struct timespec       i_mtime;
    struct timespec       i_ctime;
    unsigned int      i_blkbits;
    blkcnt_t          i_blocks;
    unsigned short      i_bytes;
    umode_t               i_mode;
    spinlock_t            i_lock;  /* i_blocks, i_bytes, maybe i_size */
    struct mutex      i_mutex;
    struct rw_semaphore   i_alloc_sem;
    const struct inode_operations  *i_op;
    const struct file_operations   *i_fop;
/*former ->i_op->default_file_ops */
    struct super_block    *i_sb;
    struct file_lock *i_flock;
    struct address_space  *i_mapping;
    struct address_space  i_data;
#ifdef CONFIG_QUOTA
    struct dquot      *i_dquot[MAXQUOTAS];
#endif
    struct list_head  i_devices;
    union {
        struct pipe_inode_info    *i_pipe;
        struct block_device    *i_bdev;
        struct cdev        *i_cdev;
    };
    int           i_cindex;

    __u32             i_generation;

#ifdef CONFIG_DNOTIFY
    unsigned long     i_dnotify_mask; /* Directory notify events */
    struct dnotify_struct *i_dnotify; /* for directory notifications */
#endif
```


华清远见 HQYJ.COM

华清远见教育集团官网：www.hqyj.com

```
#ifdef CONFIG_INOTIFY
    struct list_head inotify_watches; /* watches on this inode */
    struct mutex      inotify_mutex;    /* protects the watches list */
#endif

    unsigned long     i_state;
    unsigned long     dirtied_when;     /* jiffies of first dirtying */

    unsigned int      i_flags;

    atomic_t      i_writecount;
#ifdef CONFIG_SECURITY
    void          *i_security;
#endif
    void          *i_private;           /* fs or device private pointer */
};
```

INODE 数据结构成员说明如表 8.3 所示。

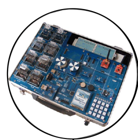
表 8.3　INODE 数据结构成员说明

字　段	含　义
i_hash	哈希表
i_list	索引节点链表
i_sb_list	超级块项链表
i_dentry	目录项链表
i_ino	节点号
i_count	引用计数
i_nlink	硬链接数
i_uid	使用者 ID
i_gid	使用者组 ID
i_rdev	实设备标识符
i_version	版本
i_size	以字节为单位的文件大小
i_size_seqcount	同步计数
i_atime	上次访问文件的时间
i_mtime	上次修改文件的时间
i_ctime	创建索引节点的时间
i_blkbits	块的位数：文件在做 I/O 时的区块大小
i_blocks	文件的块数：文件所使用的磁盘块数，一个磁盘块为 512B
i_bytes	文件中最后一个块的字节数
i_mode	文件的访问权限

华清远见

华清远见 HQYJ.COM

华清远见教育集团官网：www.hqyj.com



续表

字　段	含　义
i_lock	自旋锁
i_mutex	防止 inode 操作时互斥
i_alloc_sem	索引节点信号量
*i_op	索引节点操作表
*i_fop	默认的索引节点操作
*i_sb	指向超级块对象的指针
*i_flock	文件锁链表
*i_mapping	相关的地址映射
i_data	设备地址映射
*i_dquot[MAXQUOTAS]	节点的磁盘限额
i_devices	块设备链表
i_cindex	拥有一组此设备号的设备文件的索引
i_generation	保留
i_dnotify_mask	目录事件通知掩码
*i_dnotify	目录事件
inotify_watches	返回在该目录下的所有文件上面发生的事件
inotify_mutex	通知机制下的锁
i_state	状态标志
dirtied_when	首次修改时间
i_flags	文件系统标识
i_writecount	写计数
*i_security	安全标识
*i_private	指向私有数据的指针

那么是如何实现不同操作系统的不同读/写文件的方式呢？答案是 inode_operations 结构。这个结构是一组函数指针。inode_operations 所指向的不是针对某一个文件进行操作的函数，而是影响文件和目录布局的函数，例如，添加、删除文件和目录、跟踪符号链接等。这个结构也在 fs.h 文件中定义，其代码如下：

```
struct inode_operations {
    int (*create) (struct inode *,struct dentry *,int, struct nameidata *);
    struct dentry * (*lookup) (struct inode *,struct dentry *, struct nameidata *);
    int (*link) (struct dentry *,struct inode *,struct dentry *);
    int (*unlink) (struct inode *,struct dentry *);
    int (*symlink) (struct inode *,struct dentry *,const char *);
    int (*mkdir) (struct inode *,struct dentry *,int);
    int (*rmdir) (struct inode *,struct dentry *);
    int (*mknod) (struct inode *,struct dentry *,int,dev_t);
```


华清远见 HQYJ.COM

华清远见教育集团官网：www.hqyj.com

```
        int (*rename) (struct inode *, struct dentry *,
                struct inode *, struct dentry *);
        int (*readlink) (struct dentry *, char __user *,int);
        void * (*follow_link) (struct dentry *, struct nameidata *);
        void (*put_link) (struct dentry *, struct nameidata *, void *);
        void (*truncate) (struct inode *);
        int (*permission) (struct inode *, int);
        int (*setattr) (struct dentry *, struct iattr *);
        int (*getattr) (struct vfsmount *mnt, struct dentry *, struct kstat *);
        int (*setxattr) (struct dentry *, const char *,const void *,size_t,int);
        ssize_t (*getxattr) (struct dentry *, const char *, void *, size_t);
        ssize_t (*listxattr) (struct dentry *, char *, size_t);
        int (*removexattr) (struct dentry *, const char *);
        void (*truncate_range)(struct inode *, loff_t, loff_t);
        long (*fallocate)(struct inode *inode, int mode, loff_t offset,loff_t len);
        int (*fiemap)(struct inode *, struct fiemap_extent_info *, u64 start,
                 u64 len);
};
```

INODE_OPERATIONS 数据结构成员说明如表 8.4 所示。

表 8.4　INODE_OPERATIONS 数据结构成员说明

成　员	描　述
create	创建新节点
lookup	查找目录
link	创建硬链接
unlink	删除链接
symlink	同步链接
mkdir	创建目录
rmdir	删除目录
mknod	创建新设备节点
rename	重命名
readlink	读取链接
follow_link	解析符号链接
put_link	释放 follow_link 分配的数据结构
truncate	修改索引节点 inode 所指文件的长度
permission	确认是否允许对索引节点 inode 所指的文件进行指定模式的访问
setattr	设置属性
getattr	读取属性
setxattr	设置扩展属性
getxattr	读取扩展属性
listxattr	读取扩展属性的链表

华清远见

华清远见 HQYJ.COM 华清远见教育集团官网：www.hqyj.com

续表

成　员	描　述
removexattr	删除节点扩展属性
truncate_range	修改索引节点 inode 所指文件的长度范围
fallocate	文件块分配
fiemap	获得节点扩展信息

可以很清楚地看到，这个结构体中提供了文件创建、文件删除、文件重命名、创建和删除文件夹等各种操作的接口。不同的文件系统会提供不同的文件创建方式，只要提供不同的 create 函数即可，如 myfs_create 函数。例如，msdos 文件系统其公用索引节点的操作在 fs/msdos/namei.c 中定义如下：

```
    struct inode_operations msdos_dir_inode_operations = {
        create:        msdos_create,
        lookup:        msdos_lookup,
        unlink:        msdos_unlink,
        mkdir:         msdos_mkdir,
        rmdir:         msdos_rmdir,
        rename:        msdos_rename,
        setattr:       fat_notify_change,
};
```

类似的，如果我们自己要创建一套文件系统，如 myfs，那么需要定义属于自己文件系统的节点操作方法，代码如下：

```
    struct inode_operations myfs_inode_operations = {
        create:        myfs _create,
        lookup:        myfs _lookup,
        unlink:        myfs _unlink,
        mkdir:         myfs _mkdir,
        rmdir:         myfs _rmdir,
        rename:        myfs _rename,
…
};
```

8.3.4　VFS 的目录项

每个文件除了有一个索引节点 inode 数据结构外，还有一个目录项 dentry（directory enrty）数据结构。dentry 结构中有一个 d_inode 指针指向相应的 inode 结构。dentry 和 inode 所描述的目标不同，dentry 结构代表的是逻辑意义上的文件，所描述的是文件逻辑上的属性，因此，目录项对象在磁盘上并没有对应的映像；而 inode 结构代表的是物理意义上的文件，记录的是物理上的属性，对于 Ext2 文件系统来说，ext2_ inode 结构在磁盘上就有对应的映像。一个索引节点对象可能对应多个目录项对象。

一个有效的 dentry 结构必定有一个 inode 结构，这是因为一个目录项要么代表着一个文件，要么代表着一个目录，而目录实际上也是文件。所以，只要 dentry 结构是有效

的，则其指针 d_inode 必定指向一个 inode 结构。可是，反过来则不然，一个 inode 却可能对应着不止一个 dentry 结构；也就是说，一个文件可以有不止一个文件名或路径名。这是因为一个已经建立的文件可以被链接（link）到其他文件名。所以，在 inode 结构中有一个队列 i_dentry，凡是代表着同一个文件的所有目录项都通过其 dentry 结构中的 d_alias 域挂入相应 inode 结构中的 i_dentry 队列。

在超级块结构中，目录项被装入内存后，VFS 就把它转换为一个 dentry 结构体类型的目录项对象。目录项对象记录了存储在磁盘上的该文件系统的安装信息。引入目录项的概念主要是出于方便查找文件的目的。一个路径的各个组成部分，不管是目录还是普通的文件，都是一个目录项对象。例如，在路径/home/enty.sh 中，目录/home 和文件 enty.sh 都对应一个目录项对象。不同于前面的两个对象，目录项对象没有对应的磁盘数据结构，VFS 在遍历路径名的过程中现场将它们逐个地解析成目录项对象。目录项对象 dentry 在 include\linux\dcache.h 中定义，代码如下：

```
struct dentry {
    atomic_t d_count;
    unsigned int d_flags;           /* protected by d_lock */
    spinlock_t d_lock;              /* per dentry lock */
    struct inode *d_inode;
/* Where the name belongs to - NULL is  * negative */
    /*
     * The next three fields are touched by __d_lookup.  Place them here
     * so they all fit in a cache line.
     */
    struct hlist_node d_hash;       /* lookup hash list */
    struct dentry *d_parent;        /* parent directory */
    struct qstr d_name;

    struct list_head d_lru;         /* LRU list */
    /*
     * d_child and d_rcu can share memory
     */
    union {
        struct list_head d_child;   /* child of parent list */
        struct rcu_head d_rcu;
    } d_u;
    struct list_head d_subdirs;     /* our children */
    struct list_head d_alias;       /* inode alias list */
    unsigned long d_time;           /* used by d_revalidate */
    struct dentry_operations *d_op;
    struct super_block *d_sb;       /* The root of the dentry tree */
    void *d_fsdata;                 /* fs-specific data */
#ifdef CONFIG_PROFILING
    struct dcookie_struct *d_cookie; /* cookie, if any */
#endif
    int d_mounted;
    unsigned char d_iname[DNAME_INLINE_LEN_MIN];    /* small names */
};
```

DENTRY 数据结构成员说明如表 8.5 所示。

表 8.5　DENTRY 数据结构成员说明

成　　员	描　　述
d_count	引用计数
d_flags	高速缓存标识
d_lock	自旋锁
*d_inode	节点
d_hash	散列表表项指针
*d_parent	父目录的目录项对象
d_name	文件名
d_lru	未使用链表指针
d_u	这是一个由 d_child 和 d_rcu 构成的联合，d_child 是目录中目录项对象的链表指针，d_rcu 指向 RCU 链表指针
d_subdirs	子目录项的链表
d_alias	相关节点的链表
d_time	d_revalidate 方法使用
*d_op	目录项操作
*d_sb	超级块结构指针
*d_fsdata	文件系统相关数据
*d_cookie	指向内核配置文件使用的数据结构指针
d_mounted	挂载计数器
d_iname[DNAME_INLINE_LEN_MIN]	存放短文件名的空间

与目录项对象结构对应的操作函数结构是 dentry_operations，它目前提供了 7 个成员函数接口，如表 8.6 所示。

```
struct dentry_operations {
    int (*d_revalidate)(struct dentry *, struct nameidata *);
    int (*d_hash) (struct dentry *, struct qstr *);
    int (*d_compare) (struct dentry *, struct qstr *, struct qstr *);
    int (*d_delete)(struct dentry *);
    void (*d_release)(struct dentry *);
    void (*d_iput)(struct dentry *, struct inode *);
    char *(*d_dname)(struct dentry *, char *, int);
};
```


华清远见
HQYJ.COM
华清远见教育集团官网：www.hqyj.com

表 8.6　DENTRY_ OPERATIONS 数据结构成员说明

成　　员	描　　述
d_revalidate	在把目录条目转换成一个文件路径名之前，判定该目录条目结构是否有效
d_hash	目录项生成散列表
d_compare	VFS 调用该函数来比较 name1 和 name2 这两个文件名
d_delete	当目录项对象的 d_count 计数值等于 0 时，VFS 调用该函数
d_release	当目录项对象将要被释放时，VFS 调用该函数
d_iput	当目录项对象丢失了其相关的索引节点时，VFS 调用该函数
d_dname	保留

8.3.5　文件对象

文件对象是已打开的文件在内存中的表示，主要用于建立进程和磁盘上的文件的对应关系。它由 sys_open 创建，由 sys_close 销毁。文件对象和物理文件的关系类似于进程和程序。一个文件对应的文件对象可能不是唯一的，但是其对应的索引节点和目录项对象是唯一的。

文件对象在内核中用 file 结构表示，它在 include\linux\fs.h 中实现，代码如下：

```
struct file {
    /*
     * fu_list becomes invalid after file_free is called and queued via
     * fu_rcuhead for RCU freeing
     */
    union {
        struct list_head  fu_list;
        struct rcu_head   fu_rcuhead;
    } f_u;
    struct path       f_path;
#define f_dentry  f_path.dentry
#define f_vfsmnt  f_path.mnt
    const struct file_operations   *f_op;
    atomic_long_t     f_count;
    unsigned int      f_flags;
    fmode_t           f_mode;
    loff_t            f_pos;
    struct fown_struct    f_owner;
    unsigned int      f_uid, f_gid;
    struct file_ra_state  f_ra;

    u64          f_version;
#ifdef CONFIG_SECURITY
    void         *f_security;
#endif
    /* needed for tty driver, and maybe others */
    void         *private_data;
```


华清远见 HQYJ.COM　华清远见教育集团官网：www.hqyj.com

```
#ifdef CONFIG_EPOLL
    /* Used by fs/eventpoll.c to link all the hooks to this file */
    struct list_head f_ep_links;
    spinlock_t       f_ep_lock;
#endif /* #ifdef CONFIG_EPOLL */
    struct address_space  *f_mapping;
#ifdef CONFIG_DEBUG_WRITECOUNT
    unsigned long f_mnt_write_state;
#endif
};
```

FILE 数据结构成员说明如表 8.7 所示。

表 8.7　FILE 数据结构成员说明

字段	含义
f_u	这个联合由两个链表成员构成，主要用于更新文件。其中 RCU（Read-Copy Update）是 Linux 2.6 内核中新的锁机制，读者可以参考相关教程
f_next	指到下一个 file 结构
f_pprev	指到上一个 file 结构地址的地址
f_dentry	记录其 inode 的入口地址
f_mode	文件读取类型
f_pos	文件的读/写位置
f_count	结构的引用次数
f_flags	打开文件时指定的标识
f_owner	记录了要接收 SIGIO 和 SIGURG 的行程 ID 或行程群组 ID
private_data	系统在调用驱动程序的 open 方法前将这个指针置为 NULL。驱动程序可以将这个字段用于任意目的，也可以忽略这个字段。驱动程序可以用这个字段指向已分配的数据，但是一定要在内核释放 file 结构前的 release 方法中清除它
f_uid	文件所属用户的 ID
f_gid	文件所属组的 ID
f_security	描述安全措施或记录与安全有关的信息
f_path	它是一个 path 结构体，其中一个成员*mnt 的作用是指出该文件的已安装文件系统，另一个成员*dentry 是与文件相关的目录项对象
f_op	就是前面使用的 file_operations 结构体指针，包含与文件关联的操作

每个文件对象总是包含在下列的一个双向循环链表之中。

- “未使用”文件对象的链表：该链表既可以用做文件对象的内存高速缓存，又可以用做超级用户的备用存储器，也就是说，即使系统的动态内存用完，也允许超级用户打开文件。由于这些对象是未使用的，所以它们的 f_count 域为 NULL，该链表首元素的地址存放在变量 free_list 中，内核必须确认该链表总是至少包含 NR_RESERVED_FILES 个对象，通常设置为 10。
- “正在使用”文件对的象链表：该链表中的每个元素至少由一个进程使用，因此，各个元素的 f_count 域不会为 NULL，该链表中第一个元素的地址存放在

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

变量anon_list中。

每个进程用一个files_struct结构来记录文件描述符的使用情况，这个files_struct结构称为用户打开文件表，它是进程的私有数据。files_struct结构在include/linux/sched.h中定义。因为本章重点讲解文件系统相关内容，与进程有关的内容请读者自行研究。

与file结构相关的是file_operations结构，这个结构对于驱动程序开发非常重要，因为Linux内核把所有的设备都当成文件来打开。

```
/*
 * NOTE:
 * read, write, poll, fsync, readv, writev, unlocked_ioctl and compat_ioctl
 * can be called without the big kernel lock held in all filesystems.
 */
struct file_operations {
     struct module *owner;
     loff_t (*llseek) (struct file *, loff_t, int);
     ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
     ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
     ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long,
     loff_t);
     ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long,
     loff_t);
     int (*readdir) (struct file *, void *, filldir_t);
     unsigned int (*poll) (struct file *, struct poll_table_struct *);
     int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
     long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
     long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
     int (*mmap) (struct file *, struct vm_area_struct *);
     int (*open) (struct inode *, struct file *);
     int (*flush) (struct file *, fl_owner_t id);
     int (*release) (struct inode *, struct file *);
     int (*fsync) (struct file *, struct dentry *, int datasync);
     int (*aio_fsync) (struct kiocb *, int datasync);
     int (*fasync) (int, struct file *, int);
     int (*lock) (struct file *, int, struct file_lock *);
     ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
     unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long,
     unsigned long, unsigned long);
     int (*check_flags)(int);
     int (*dir_notify)(struct file *filp, unsigned long arg);
     int (*flock) (struct file *, int, struct file_lock *);
     ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *,
     size_t, unsigned int);
     ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *,
     size_t, unsigned int);
     int (*setlease)(struct file *, long, struct file_lock **);
};
```

FILE_OPERATIONS数据结构成员说明如表8.8所示。

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com



表 8.8　FILE_OPERATIONS 数据结构成员说明

字　段	含　义
owner	指向拥有这个结构的模块的指针
llseek	改变文件中的当前读/写位置，并且新位置作为（正的）返回值.
read	从设备中读取数据
write	向设备中写入数据
aio_read	初始化一个异步读——可能在函数返回前不结束的读操作
aio_write	初始化一个异步写——可能在函数返回前不结束的写操作
Readdir	它用来读取目录，并且仅对文件系统有用
Poll	查询对一个或多个文件描述符的读或写是否会阻塞
ioctl	ioctl 系统调用提供了发出设备特定命令的方法
unlocked_ioctl	未上锁的 ioctl
compat_ioctl	兼容的 ioctl
mmap	用来请求将设备内存映射到进程的地址空间
open	打开设备时调用
flush	在进程关闭它的设备文件描述符的复制时调用
release	在文件结构被释放时引用这个操作
fsync	fsync 系统调用的后端，用户调用来刷新任何挂着的数据
aio_fsync	fsync 方法的异步版本
fasync	通知设备的 FASYNC 标识的改变
lock	实现文件锁
sendpage	实现 sendfile 系统调用的读
get_unmapped_area	获取未映射区域
check_flags	检查标识
dir_notify	在应用程序使用 fcntl 来请求目录改变通知时调用
flock	flock 系统调用的后端
splice_write	向设备中写入数据
splice_read	从设备中读取数据

华清远见

8.3.6　主要数据结构间的关系

前面介绍了超级块对象、索引节点对象、文件对象及目录项对象的数据结构。下面给出这些数据结构之间的联系。超级块是对一个文件系统的描述；索引节点是对一个文件物理属性的描述；而目录项是对一个文件逻辑属性的描述。除此之外，文件与进程之间的关系是由另外的数据结构来描述的。一个进程所处的位置是由 fs_struct 来描述的，

华清远见 HQYJ.COM

华清远见教育集团官网：www.hqyj.com

而一个进程打开的文件是由 files_struct 来描述的，整个系统所打开的文件是由 file 结构来描述。如图 8.7 所示为这些数据结构之间的关系。

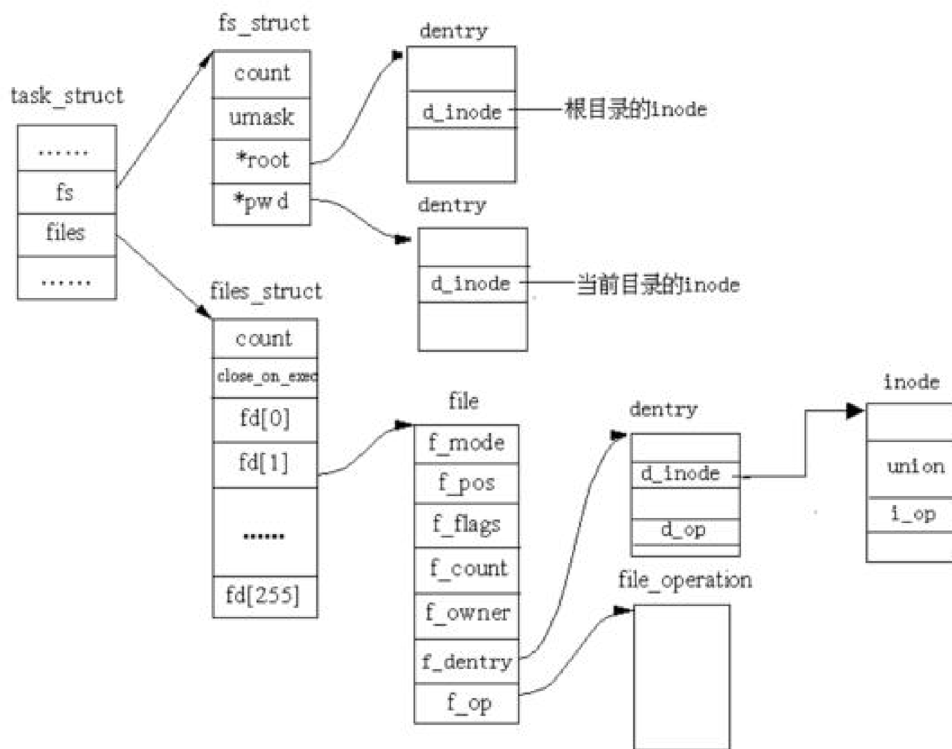


图 8.7　与进程联系的文件结构的关系示意图

8.4　文件系统注册与卸载

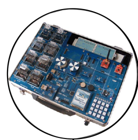


8.4.1　和文件系统相关的数据结构

根据文件系统所在的物理介质和数据在物理介质上的组织方式来区分不同的文件系统类型。file_system_type 结构用于描述各种特定文件系统类型，如 Ext3 或 VFAT。每种文件系统，不管有多少个实例安装（或没有安装）到系统中，都只有一个 file_system_type 结构。

file_system_type 在 include\linux\fs.h 中定义，代码如下：

```
struct file_system_type {
    const char *name;
    int fs_flags;
```




```
    int (*get_sb) (struct file_system_type *, int,
              const char *, void *, struct vfsmount *);
    void (*kill_sb) (struct super_block *);
    struct module *owner;
    struct file_system_type * next;
    struct list_head fs_supers;

    struct lock_class_key s_lock_key;
    struct lock_class_key s_umount_key;

    struct lock_class_key i_lock_key;
    struct lock_class_key i_mutex_key;
    struct lock_class_key i_mutex_dir_key;
    struct lock_class_key i_alloc_sem_key;
};
```

FILE_SYSTEM_TYPE 数据结构成员说明如表 8.9 所示。

表 8.9　FILE_SYSTEM_TYPE 数据结构成员说明

字　　段	含　　义
*name	文件系统的名字
fs_flags	文件系统类型标识
*get_sb	访问超级块
*kill_sb	删除超级块时调用
*owner	指向拥有这个结构的文件系统的指针
* next	链表中的下一个文件系统类型
fs_supers	具有同一种文件系统类型的超级块对象链表
s_lock_key	自旋锁相关的几个成员
s_umount_key	
i_lock_key	
i_mutex_key	
i_mutex_dir_key	
i_alloc_sem_key	

以 cifs 文件系统为例，看一下 file_system_type 在内核中的使用方法。

```
struct file_system_type cifs_fs_type = {
    .owner = THIS_MODULE,
    .name = "cifs",
    .get_sb = cifs_get_sb,
    .kill_sb = kill_anon_super,
    /* .fs_flags */
};
```

如图 8.8 所示为已经在系统注册的文件系统形成的链表。

华清远见 HQYJ.COM　华清远见教育集团官网：www.hqyj.com

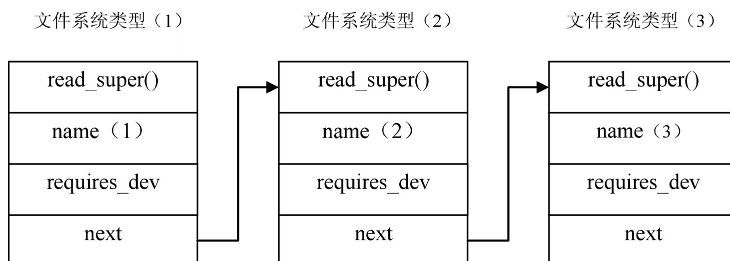
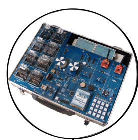


图 8.8　已注册的文件系统链表

还有一个数据结构是 vfsmount。Linux 对系统中已经安装的文件系统都通过这个结构描述。各 vfsmount 构成一棵树，vfsmount 树中的每个节点对应且仅对应一棵 dentry 树。vfsmount 在 include\linux\mount.h 中定义，代码如下：

```
struct vfsmount {
     struct list_head mnt_hash;
     struct vfsmount *mnt_parent;   /* fs we are mounted on */
     struct dentry *mnt_mountpoint; /* dentry of mountpoint */
     struct dentry *mnt_root;       /* root of the mounted tree */
     struct super_block *mnt_sb;    /* pointer to superblock */
     struct list_head mnt_mounts;   /* list of children, anchored here */
     struct list_head mnt_child;    /* and going through their mnt_child */
     int mnt_flags;
     /* 4 bytes hole on 64bits arches */
     const char *mnt_devname;       /* Name of device e.g. /dev/dsk/hda1 */
     struct list_head mnt_list;
     struct list_head mnt_expire;   /* link in fs-specific expiry list */
     struct list_head mnt_share;    /* circular list of shared mounts */
     struct list_head mnt_slave_list;/* list of slave mounts */
     struct list_head mnt_slave;    /* slave list entry */
     struct vfsmount *mnt_master;   /* slave is on master->mnt_slave_list */
     struct mnt_namespace *mnt_ns;  /* containing namespace */
     int mnt_id;                    /* mount identifier */
     int mnt_group_id;              /* peer group identifier */
     /*
      * We put mnt_count & mnt_expiry_mark at the end of struct vfsmount
      * to let these frequently modified fields in a separate cache line
      * (so that reads of mnt_flags wont ping-pong on SMP machines)
      */
     atomic_t mnt_count;
     int mnt_expiry_mark;       /* true if marked for expiry */
     int mnt_pinned;
     int mnt_ghosts;
     /*
      * This value is not stable unless all of the mnt_writers[] spinlocks
      * are held, and all mnt_writer[]s on this mount have 0 as their ->count
      */
     atomic_t __mnt_writers;
};
```


华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com



VFSMOUNT 数据结构成员说明如表 8.10 所示。

表 8.10　VFSMOUNT 数据结构成员说明

字　段	含　义
mnt_hash	挂载的文件系统结构
*mnt_parent	所挂载到的父文件系统结构
*mnt_mountpoint	挂载点
*mnt_root	文件系统根目录的目录条目结构指针
*mnt_sb	挂载文件系统的超级块
mnt_mounts	已挂载的文件系统描述符结构链表
mnt_child	已挂载的子文件系统描述符
mnt_flags	挂载标识
*mnt_devname	指向文件系统类型结构的 name 字段
mnt_list	挂载到自己的命名空间结构的 list 链表中
mnt_expire	过期标识
mnt_share	共享标识
mnt_slave_list	slave 挂载点列表
mnt_slave	slave 链表
*mnt_master	主链表
*mnt_ns	命名空间
mnt_id	挂载点 id
mnt_group_id	挂载点组 id
mnt_count	引用计数器
mnt_expiry_mark	过期标识
mnt_pinned	挂载点被钉标识
mnt_ghosts	克隆标识
__mnt_writers	同步

华清远见

8.4.2　文件系统类型注册函数

文件系统类型注册是把具体的文件系统类型的 file_system_type 结构对象链接到系统已经注册文件系统类型链表中。使用的注册函数为 register_filesystem()，该函数在 fs\filesystems.c 中实现，代码如下：

```
int register_filesystem(struct file_system_type * fs)
{
    int res = 0;
    struct file_system_type ** p;
```


华清远见 HQYJ.COM

华清远见教育集团官网：www.hqyj.com

```
        BUG_ON(strchr(fs->name, '.'));
        if (fs->next)
            return -EBUSY;
        INIT_LIST_HEAD(&fs->fs_supers);
        write_lock(&file_systems_lock);
        p = find_filesystem(fs->name, strlen(fs->name));
        if (*p)
            res = -EBUSY;
        else
            *p = fs;
        write_unlock(&file_systems_lock);
        return res;
    }
```

8.4.3　挂载文件系统

挂载文件系统的过程是为文件系统创建对应的 VFS 数据结构对象，并且把这些数据结构对象链接到系统中各自对应的全局链表中。挂载文件系统是通过 mount 系统调用完成的。mount 调用的内核实现函数是 sys_mount()，它在 fs\namespace.c 中实现，代码如下：

```
    asmlinkage long sys_mount(char __user * dev_name, char __user * dir_name,
              char __user * type, unsigned long flags, void __user * data)
    {
        int retval;
        unsigned long data_page;
        unsigned long type_page;
        unsigned long dev_page;
        char *dir_page;

        retval = copy_mount_options(type, &type_page);
        if (retval < 0)
            return retval;

        dir_page = getname(dir_name);
        retval = PTR_ERR(dir_page);
        if (IS_ERR(dir_page))
            goto out1;

        retval = copy_mount_options(dev_name, &dev_page);
        if (retval < 0)
            goto out2;

        retval = copy_mount_options(data, &data_page);
        if (retval < 0)
            goto out3;

        lock_kernel();
        retval = do_mount((char *)dev_page, dir_page, (char *)type_page,
                    flags, (void *)data_page);
        unlock_kernel();
```


华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com




```
    free_page(data_page);

out3:
    free_page(dev_page);
out2:
    putname(dir_page);
out1:
    free_page(type_page);
    return retval;
}
```

其中的各参数说明如下。

- dev_name：用户空间的存放文件系统的设备名。
- dir_name：用户空间的挂载文件系统的目录名。
- type：文件系统类型。
- flags：挂载标志。
- data：传递给超级块读取函数的参数指针。

8.4.4 文件系统卸载

Linux 文件系统可以根据需要随时卸载，从而实现文件存储空间的动态扩充。卸载文件系统的过程和安装过程相反。

卸载文件系统的命令是 unmount，卸载时，首先验证被卸载文件系统是否可以卸载，如果该文件系统中的文件正被使用，那么在 VFS inode 节点的缓存中就会有来自该文件系统的 VFS inode 节点。检测程序通过在 inode 节点表中查找来自该文件系统所在设备的 inode 节点可以检测出这个问题。如果相应的节点标识为“被修改过”，则该文件系统不能被卸载。当一切检查完成后，则释放对应的 VFS 超级块和安装点，最后为装配操作建立的 vfsmount 数据结构与 vfsmntlist 解除链接并释放掉，从而卸下该文件系统。

8.5 本章习题



1．什么是文件目录？
2．什么是文件系统？
3．什么是超级块？在 Linux 内核中是如何实现的？
4．什么是节点？在 Linux 内核中是如何实现的？

华清远见 HQYJ.COM 华清远见教育集团官网：www.hqyj.com

华清远见 HQYJ.COM

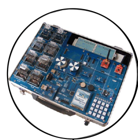


第 9 章 设备管理

外部设备是计算机系统最要的组成部分，是计算机主机与外部环境进行交互的手段。操作系统通过软件对它们进行高度抽象，屏蔽外部设备的各种差异，为用户提供一个友好的操作界面。I/O 设备是计算机最基本的 3 个物质基础之一，现代计算机系统都配有种类繁多的 I/O 设备，功能各不相同，且特性和操作方法也存在很大的差异，因此，用于控制设备的 I/O 系统成为系统中最繁杂且与硬件最紧密相关的部分。设备管理必须对这些繁多的设备进行管理和控制，使用户能够简单、方便、高效、统一地使用各种设备。

华清远见

华清远见 HQYJ.COM 华清远见教育集团官网：www.hqyj.com



9.1 设备及设备管理的功能



在计算机系统中，除 CPU 和内存外，其他大部分硬件设备都称为外部设备，包括常用的硬盘、光驱、输入/输出、终端等。这些设备种类非常多，操作方式也不相同，因此，操作系统的设备管理非常复杂。我们首先了解设备的分类。

9.1.1 设备分类

可以从不同的角度对设备进行分类。

- 按系统和用户分可分为：系统设备、用户设备。
- 按输入/输出传送方式分（UNIX 或 Linux 操作系统）可分为：字符设备、块设备。
- 按资源特点分可分为：独享设备、共享设备、虚拟设备。
- 按设备硬件物理特性分可分为：顺序存取设备、直接存取设备。
- 按设备使用分可分为：物理设备、逻辑设备、伪设备。
- 按数据组织分可分为：块设备、字符设备。
- 按数据传输率分可分为：低速设备、中速设备、高速设备。

根据设备的用途，可以把设备分为存储设备与输入/输出设备两大类。

存储设备是指用来进行数据存储的设备，计算机的存储器分为主存储器（内存）和辅助存储器（外存），外部存储器就是一种典型的存储类型的设备，例如，硬盘、软盘、CD、U 盘、移动硬盘等。

输入/输出设备是主机从外界接收信息或向外界发送信息的媒介。输入设备是计算机用来从外界接收信息的设备，例如，鼠标、键盘、扫描仪等；输出设备是计算机把处理后的信息发向外界的设备，例如，打印机、显示器等。这种设备以每次一个字符的方式发送数据，因此称为字符设备。

华清远见

9.1.2 设备管理

设备管理的主要功能是分配和回收外部设备，以及控制外部设备按用户程序的要求进行操作等。对于非存储型外部设备，如打印机、显示器等，它们可以直接作为一个设备分配给一个用户程序，使用完毕后回收以便给另一个需求的用户使用。对于存储型的外部设备，如磁盘、磁带等，则提供存储空间给用户，用来存放文件和数据。存储性外部设备的管理与信息管理是密切结合的。

设备管理的目标有两个：一个是按用户需求提出的要求接入外部设备，系统按一定算法分配和管理控制，而用户不必关心设备的实际地址和控制指令；另外，要尽量提高

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

输入/输出设备的利用率，例如，发挥主机与外设及外设与外设之间的真正并行工作能力。主要利用的技术有中断技术、DMA 技术、通道技术、缓冲技术。

设备管理的任务也比较繁重，以下几项包括：

- 动态掌握并记录设备的状态。
- 分配设备和释放。
- 对输入/输出缓冲区进行管理。
- 控制和实现真正的输入/输出操作。
- 提供设备使用的用户接口。
- 在一些较大系统中实现虚拟设备技术。

通道（Channel）是计算机系统中能够独立完成输入/输出操作的硬件装置，又称“输入/输出处理器”。 虽然在 CPU 与 I/O 设备之间增加了设备控制器，但 CPU 的负担仍很重。为此，在 CPU 和设备控制器之间又增设了 I/O 通道。其目的是使一些原来由 CPU 处理的 I/O 任务转由通道来承担，从而把 CPU 从繁杂的 I/O 任务中解脱出来。

9.2 I/O 内核子系统



CPU 与外部设备存储器的连接和数据交换都需要通过接口设备来实现。CPU 与外部设备存储器的连接被称为 I/O 接口，而数据交换则称为存储器接口。存储器通常在 CPU 的同步控制下工作。接口电路比较简单，而 I/O 设备品种繁多，其相应的接口电路也各不相同。使用设备完成输入/输出的过程，就是主机与外部设备之间数据传送的过程。I/O 管理最主要的任务是：完成用户提出的 I/O 请求、提高 I/O 速率及提高设备的利用率，并能为更高层的进程方便地使用这些设备提供手段。

9.2.1 I/O 系统的基本功能

为了满足系统和用户的要求，I/O 系统应具有以下几个方面的基本功能。

功能一：隐藏物理设备的细节

I/O 设备的类型很多，并且彼此间在多方面都有差异，例如，它们在接收和产生数据的速度、方向、数据表现形式，以及可靠性等方面。这是一种硬件设备，其中包含若干个用于存放控制命令的寄存器和存放参数的寄存器。用户通过这些命令和参数，可以控制外部设备执行所要求的操作。为使处理器可以像访问存储器那样来访问外部设备，控制器必须提供一些互不冲突、能按地址访问，并能以数字信号进行数据传送信息的寄存器。根据需要，外部设备通常包含 4 组寄存器：状态寄存器、控制寄存器、数据输入寄存器和数据输出寄存器。这些寄存器都有自己的地址，每个地址称为一个端口。这 4 组寄存器的作用分别如下：

华清远见 HQYJ.COM 华清远见教育集团官网：www.hqyj.com

- 状态寄存器用来向处理器提供外部设备的工作状态。
- 控制寄存器用来管理外部设备的工作模式。
- 数据输入寄存器和数据输出寄存器都是数据缓存寄存器，即在外设与处理器传递数据时的数据暂存器。

为访问这些外部设备，系统必须为外设的这些寄存器分配地址空间，外设所占用的地址空间称为 I/O 空间。

功能二：与设备的无关性

一方面，用户不仅可以使用抽象的 I/O 命令，还可以使用抽象的逻辑设备名来使用设备；另一方面，也可以有效地提高操作系统的可移植性和易适应性。对于操作系统本身来说，应该允许在不需要将整个操作系统进行重新编译的情况下，增添新的设备驱动程序，从而做到即插即用。

功能三：提高处理器与 I/O 设备的利用率

在一般的系统中，许多 I/O 设备间是相互独立的，但是能够并行操作，处理器与设备之间也能并行操作。因此，I/O 系统要尽可能地让处理器和 I/O 设备并行操作，以提高它们的利用率。为此，一方面要求处理器能快速响应用户的 I/O 请求，使 I/O 设备尽快地运行起来；另一方面，也应尽量减少在每个 I/O 设备运行时，处理器的干预时间。

功能四：对 I/O 设备进行控制

对 I/O 设备进行控制是驱动程序的功能。目前，对 I/O 设备有 3 种控制方式：采用轮询的可编程 I/O 方式、采用中断的可编程 I/O 方式和直接存储器访问方式。究竟采用哪种控制方式，与 I/O 设备的传输速率、传输的数据单位等因素有关。

功能五：能确保对设备的正确共享

从设备的共享属性上，可将系统中的设备分为以下两类。

- 独占设备：进程应互斥地访问这些设备，即系统一旦把这类设备分配给了某个进程后，便由该进程独占，直到进程结束完全释放。
- 共享设备：是指在一段时间内允许多个进程同时访问的设备。典型的共享设备是磁盘。

功能六：错误处理

大多数设备都包括较多的机械和电气部分，运行时容易出现错误和故障。从处理的角度，可将错误分为临时性错误和持久性错误。对于临时性错误，可通过重试操作来纠正，只有在发生了持久性错误时，才需要向上层报告。

9.2.2 I/O 空间

为了访问某些外部设备，系统必须为外设的寄存器分配地址空间，外设所占用的地址空间称为 I/O 空间。

对于 I/O 空间的处理，目前有两种方式：一种是存储器统一编址，即在整个内存空间中划出一个范围作为 I/O 空间，这种方式的特点是处理器没有独立的 I/O 指令，而是

华清远见 HQYJ.COM　华清远见教育集团官网：www.hqyj.com

把外部设备的寄存器作为存储单元来对待；另一种方式是处理器具有单独的 I/O 指令，所以在这种方式中，I/O 地址空间是独立编址的。不管是统一编址还是独立编址，从逻辑上看，外设控制器的寄存器就是一种存储装置，它可以向处理器提供数据，也可以接受处理器的数据，即处理器可以对它们进行读/写操作。

有些体系结构的 CPU（如 PowerPC、m68k 等）通常只实现一个物理地址空间。在这种情况下，外设 I/O 端口的物理地址就被映射到 CPU 的单一物理地址空间中，而成为内存的一部分。此时，CPU 可以像访问一个内存单元那样访问外设 I/O 端口，而不需要设立专门的外设 I/O 指令。这就是所谓的“内存映射方式（Memory-mapped）”。

而另外一些体系结构的 CPU（如 X86）则为外设专门实现了一个单独的地址空间，称为“I/O 地址空间”或者“I/O 端口空间”。这是一个与 CPU 的 RAM 物理地址空间不同的地址空间，所有外设的 I/O 端口均在这一空间中进行编址。CPU 通过设立专门的 I/O 指令（如 X86 的 IN 和 OUT 指令）来访问这一空间中的地址单元（即 I/O 端口）。这就是所谓的“I/O 映射方式（I/O-mapped）”。与 RAM 物理地址空间相比，I/O 地址空间通常都比较小，如 X86 CPU 的 I/O 空间只有 64KB（0-0xffff）。这是“I/O 映射方式”的一个主要缺点。

通常，操作系统使用一个数据结构来描述端口资源，Linux 系统用来描述各种 I/O 资源的数据结构是 resource，它在 include\linux\ioport.h 中定义，代码如下：

```
/*
 * Resources are tree-like, allowing
 * nesting etc..
 */
struct resource {
     resource_size_t start;
     resource_size_t end;
     const char *name;
     unsigned long flags;
     struct resource *parent, *sibling, *child;
};
```

其中，各参数含义如下。

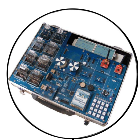
- start 和 end：表示资源的起始物理地址和终止物理地址。它们确定了资源的范围，也就是一个闭区间[start,end]。
- *name：是指向此资源的名称。
- flags：描述资源属性的标志。
- *parent, *sibling, *child：分别指向父、兄弟和子资源的指针。

常用属性标志位的定义在 ioport.h 文件中，代码如下：

```
/*
 * IO resources have these defined flags.
 */
#define IORESOURCE_BITS        0x000000ff   /* Bus-specific bits */

#define IORESOURCE_TYPE_BITS   0x00000f00   /* Resource type */
```


华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com




```
#define IORESOURCE_IO       0x00000100
#define IORESOURCE_MEM          0x00000200
#define IORESOURCE_IRQ          0x00000400
#define IORESOURCE_DMA          0x00000800

#define IORESOURCE_PREFETCH     0x00001000    /* No side effects */
#define IORESOURCE_READONLY     0x00002000
#define IORESOURCE_CACHEABLE    0x00004000
#define IORESOURCE_RANGELENGTH 0x00008000
#define IORESOURCE_SHADOWABLE  0x00010000

#define IORESOURCE_SIZEALIGN    0x00020000    /* size indicates alignment */
#define IORESOURCE_STARTALIGN  0x00040000    /* start field is alignment */

#define IORESOURCE_DISABLED     0x10000000
#define IORESOURCE_UNSET    0x20000000
#define IORESOURCE_AUTO         0x40000000
#define IORESOURCE_BUSY         0x80000000    /* Driver has marked this resource
busy */
```

Linux 在初始化时，登记系统所有的外设 I/O 资源，并以树形结构将每一种资源组织起来。树中的每一个节点都是一个 resource 结构变量，而树的根节点则描述了整个资源空间。资源的登记是通过 request_resource 函数完成的。这个函数在 kernel\resource.c 中实现，代码如下：

```
/**
 * request_resource - request and reserve an I/O or memory resource
 * @root: root resource descriptor
 * @new: resource descriptor desired by caller
 *
 * Returns 0 for success, negative error code on error.
 */
int request_resource(struct resource *root, struct resource *new)
{
    struct resource *conflict;

    write_lock(&resource_lock);
    conflict = __request_resource(root, new);
    write_unlock(&resource_lock);
    return conflict ? -EBUSY : 0;
}
```

完成 I/O 资源的释放的函数是 release_resource()。该函数只有一个参数——指针 old，它指向所要释放的资源。源代码如下：

```
/**
 * release_resource - release a previously reserved resource
 * @old: resource pointer
 */
int release_resource(struct resource *old)
{
    int retval;
```


华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

```
        write_lock(&resource_lock);
        retval = __release_resource(old);
        write_unlock(&resource_lock);
        return retval;
    }
```

出于竞争的考虑，进行真正的请求（释放）资源之前，一定要进行锁保护。实际上，无论是 request_resource 还是 release_resource，都用到了自旋锁，在保证不会被干扰的情况下，进而调用__request_resource 和__release_resource，完成真正的资源请求和释放操作。

Linux 还实现了用于 I/O 资源分配、释放和检查的函数。它们都定义在 kernel/resource.c 文件中。__request_region 用来申请一段区域，代码如下：

```
    struct resource * __request_region(struct resource *parent,
                        resource_size_t start, resource_size_t n,
                        const char *name)
    {
        struct resource *res = kzalloc(sizeof(*res), GFP_KERNEL);

        if (!res)
            return NULL;

        res->name = name;
        res->start = start;
        res->end = start + n - 1;
        res->flags = IORESOURCE_BUSY;

        write_lock(&resource_lock);

        for (;;) {
            struct resource *conflict;

            conflict = __request_resource(parent, res);
            if (!conflict)
                break;
            if (conflict != parent) {
                parent = conflict;
                if (!(conflict->flags & IORESOURCE_BUSY))
                    continue;
            }

            /* Uhhuh, that didn't work out.. */
            kfree(res);
            res = NULL;
            break;
        }
        write_unlock(&resource_lock);
        return res;
    }
```

__release_region()实现在一个父资源节点 parent 中释放给定范围的 I/O Region。实际上该函数的实现思想与__release_resource()相类似。其源代码如下：




```
void __release_region(struct resource *parent, resource_size_t start,
            resource_size_t n)
{
    struct resource **p;
    resource_size_t end;

    p = &parent->child;
    end = start + n - 1;

    write_lock(&resource_lock);

    for (;;) {
        struct resource *res = *p;

        if (!res)
            break;
        if (res->start <= start && res->end >= end) {
            if (!(res->flags & IORESOURCE_BUSY)) {
                p = &res->child;
                continue;
            }
            if (res->start != start || res->end != end)
                break;
            *p = res->sibling;
            write_unlock(&resource_lock);
            kfree(res);
            return;
        }
        p = &res->sibling;
    }

    write_unlock(&resource_lock);

    printk(KERN_WARNING "Trying to free nonexistent resource "
        "<%016llx-%016llx>\n", (unsigned long long)start,
        (unsigned long long)end);
}
```

__check_region 函数用于检查指定的地址区间是否已经被占用，其源代码如下：

```
int __check_region(struct resource *parent, resource_size_t start,
            resource_size_t n)
{
    struct resource * res;

    res = __request_region(parent, start, n, "check-region");
    if (!res)
        return -EBUSY;
    release_resource(res);
    kfree(res);
    return 0;
}
```


华清远见 HQYJ.COM 华清远见教育集团官网：www.hqyj.com

9.2.3　I/O 控制方式

使用设备完成输入/输出的过程，就是主机与外部设备之间数据传送的过程。设备管理的主要任务之一是控制设备与内存或 CPU 之间的数据传送。本节介绍几种常用的传输控制方式。

1．轮询

对 I/O 设备的程序轮询的方式，是早期的计算机系统对 I/O 设备的一种管理方式。它定时对各种设备轮流询问一遍有无处理要求。轮流询问之后，有要求的，则加以处理。在处理 I/O 设备的要求之后，处理器返回继续工作。尽管轮询需要时间，但轮询比 I/O 设备的速度要快得多，所以一般不会发生不能及时处理的问题。当然，再快的处理器，能处理的输入/输出设备的数量也是有一定限度的。而且，程序轮询毕竟占据了 CPU 相当一部分处理时间，因此程序轮询是一种效率较低的方式，在现代计算机系统中已很少应用。

2．中断

在 I/O 设备中断方式下，中央处理器与 I/O 设备之间数据的传输步骤如下：

（1）当某个进程需要数据时，发出指令启动输入/输出设备准备数据。

（2）在进程发出指令启动设备之后，该进程放弃处理器，等待相关 I/O 操作完成。此时，进程调度程序会调度其他就绪进程使用处理器。

（3）当 I/O 操作完成时，输入/输出设备控制器通过中断请求线向处理器发出中断信号，处理器收到中断信号之后，转向预先设计好的中断处理程序，对数据传送工作进行相应的处理。

（4）获得数据的进程转入就绪状态。在随后的某个时刻，进程调度程序会选中该进程继续工作。

I/O 设备中断方式使处理器的利用率提高，且能支持多道程序和 I/O 设备的并行操作。不过，中断方式仍然存在一些问题。首先，现代计算机系统通常配置有各种各样的输入/输出设备。如果这些 I/O 设备都通过中断处理方式进行并行操作，那么中断次数的急剧增加会造成 CPU 无法响应中断和出现数据丢失现象。其次，如果 I/O 控制器的数据缓冲区比较小，在缓冲区装满数据之后将会发生中断。那么，在数据传送过程中，发生中断的机会较多，这将耗去大量的 CPU 处理时间。

3．DMA（直接内存存取）

直接内存存取技术是指，数据在内存与 I/O 设备间直接进行成块传输。DMA 有两个技术特征，首先是直接传送，其次是块传送。所谓直接传送，即在内存与 I/O 设备间传送一个数据块的过程中，不需要 CPU 的任何中间干涉，只需要 CPU 在过程开始时向

华清远见

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

设备发出“传送块数据”的命令，然后通过中断来得知过程是否结束和下次操作是否准备就绪。

一个完整的 DMA 过程应包括：初始化、DMA 请求、DMA 响应、DMA 传输、DMA 结束 5 个阶段。DMA 工作过程如图 9.1 所示。

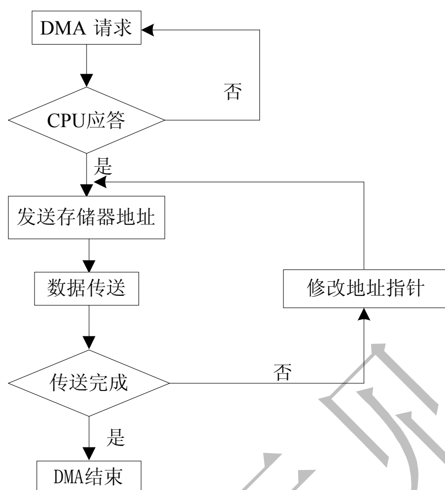


图 9.1　DMA 工作过程

在 DMA 方式中，由于 I/O 设备直接同内存发生成块的数据交换，因此 I/O 效率比较高。由于 DMA 技术可以提高 I/O 效率，因此在现代计算机系统中，得到了广泛的应用。许多输入/输出设备的控制器，特别是块设备的控制器，都支持 DMA 方式。

4．通道

输入/输出通道是一个独立于 CPU 的、专门管理 I/O 的处理器，它控制设备与内存直接进行数据交换。它有自己的通道指令，这些通道指令由 CPU 启动，并在操作结束时向 CPU 发出中断信号。输入/输出通道控制是一种以内存为中心，实现设备和内存内直接交换数据的控制方式。在通道方式中，数据的传送方向、存放数据的内存起始地址及传送的数据块长度等都由通道来进行控制。另外，通道控制方式可以做到一个通道控制多台设备与内存进行数据交换。因而，通道方式进一步减轻了 CPU 的工作负担，增加了计算机系统的并行工作程度。

9.3 Linux 设备驱动程序



设备驱动程序是 Linux 内核的重要组成部分。如操作系统的其他部分一样，驱动程序在一个高优先级的环境下工作，如果发生错误，可能会引发严重的问题。设备驱动程

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

序控制了操作系统和硬件设备之间的交互。设备驱动程序与外部设备之间的关系如图 9.2 所示。

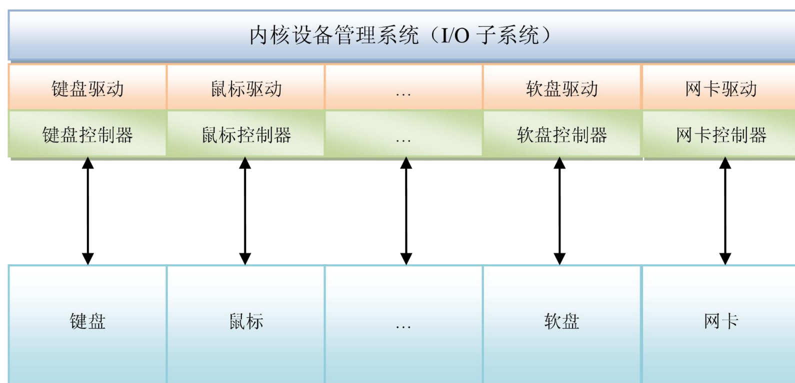


图 9.2　设备驱动程序与外部设备之间的关系

驱动程序大致分为几个主要组成部分：

- 自动配置和初始化子程序，负责检测所要驱动的硬件设备是否存在和是否能正常工作。如果该设备正常，则对这个设备及其相关的、设备驱动程序需要的软件状态进行初始化。这部分驱动程序只有在初始化时被调用一次。
- 完成用户进程请求的程序，即永恒进程对设备的操控部分。
- 设备中断服务程序，通常分为上半部和下半部。

9.3.1　设备管理

设备管理包括两个方面的内容：一是如何在系统中登记注册设备及其驱动程序，以使系统知道设备的存在及状态；二是当进程需要使用外部设备时，采用哪种方式将设备及其驱动程序提交给进程。其核心内容就是设备驱动程序的管理。

从进程的角度看，外部设备的驱动程序就是一组包括中断服务程序的操作函数集合。其中，对于外设的中断管理，我们无须操心，因为计算机系统中已经存在一个完善的中断管理系统。所以，重点要关心如何向应用进程提供那些供进程调用的操作函数及管理方法。

9.3.2　Linux 字符设备

字符设备（Character Device）和普通文件之间的主要区别是：普通文件可以来回读/写，而大多数字符设备仅仅是数据通道，只能顺序读/写。但是不能完全排除字符设备模拟普通文件读/写过程的可能性。字符设备是 Linux 最简单的设备，可以像文件一样访

华清远见
HQYJ.COM
华清远见教育集团官网：www. hqyj. com

问，如图 9.3 所示。应用程序使用标准系统调用打开、读取、写入和关闭，完全好像这个设备是一个普通文件一样，甚至连接一个 Linux 系统上网的 PPP 守护进程使用的 Modem（调制解调器）也是这样的。初始化字符设备时，它的设备驱动程序向 Linux 登记，并在字符设备向量表中增加一个 device_struct 数据结构条目，这个设备的主设备标识符（如对于 tty 设备的主设备标识符是 4）用做这个向量表的索引。一个设备的主设备标识符是固定的。chr devs 向量表中的每一个条目，一个 device_struct 数据结构，包括两个元素：一个登记的设备驱动程序的名称的指针和一个指向一组文件操作的指针。这块文件操作本身位于这个设备的字符设备驱动程序中，每一个都处理特定的文件操作，如打开、读、写和关闭。/proc/devices 中字符设备的内容来自 chrdevs 向量表，可参见 include/linux/major.h。

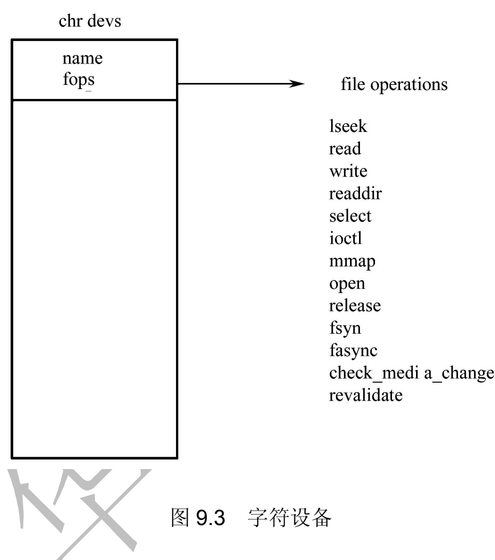


图 9.3　字符设备

当代表一个字符设备（如/dev/cua0）的字符特殊文件被打开时，核心必须做一些事情，从而匹配正确的字符设备驱动程序的文件操作例程。与普通文件或目录一样，每一个设备特殊文件都用 VFS inode 表达。这个字符特殊文件的 VFS inode（实际上所有的设备特殊文件）都包括设备的 major 和 minor 标识符。这个 VFS inode 由底层的文件系统（如 Ext2）在查找这个设备特殊文件时根据实际的文件系统创建。

参见 fs/ext2/inode.c->ext2_read_inode( )。

注册字符设备的函数是 register_chrdev()。它在 linux/fs/char_dev.c 中实现，代码如下：

```
int register_chrdev(unsigned int major, const char *name,
            const struct file_operations *fops)
{
    struct char_device_struct *cd;
    struct cdev *cdev;
    char *s;
    int err = -ENOMEM;
```


华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

```
        cd = __register_chrdev_region(major, 0, 256, name);
        if (IS_ERR(cd))
            return PTR_ERR(cd);

        cdev = cdev_alloc();
        if (!cdev)
            goto out2;

        cdev->owner = fops->owner;
        cdev->ops = fops;
        kobject_set_name(&cdev->kobj, "%s", name);
        for (s = strchr(kobject_name(&cdev->kobj),'/'); s; s = strchr(s, '/'))
            *s = '!';
        err = cdev_add(cdev, MKDEV(cd->major, 0), 256);
        if (err)
            goto out;

        cd->cdev = cdev;

        return major ? 0 : cd->major;
    out:
        kobject_put(&cdev->kobj);
    out2:
        kfree(__unregister_chrdev_region(cd->major, 0, 256));
        return err;
    }
```

9.3.3 Linux 块设备

块设备（Block Device）是文件系统的物质基础，它也支持像文件一样被访问。这种为打开的块特殊文件提供正确的文件操作组的机制和字符设备的十分相似。Linux 用 blkdevs 向量表维护已经登记的块设备文件。它像 chrdevs 向量表一样，使用设备的主设备号作为索引。它的条目也是 device_struct 数据结构。与字符设备不同，块设备进行分类时，SCSI 是其中一类，而 IDE 是另一类。类向 Linux 内核登记并向核心提供文件操作。一种块设备类的设备驱动程序向这种类提供和类相关的接口。例如，SCSI 设备驱动程序必须向 SCSI 子系统提供接口，让 SCSI 子系统用来对核心提供这种设备的文件操作。参见 fs/devices.c。

每一个块设备驱动程序必须提供普通的文件操作接口和对于 buffer Cache 的接口。每一个块设备驱动程序填充 blk_dev 向量表中的 blk_dev_struct 数据结构。这个向量表的索引还是设备的主设备号。这个 blk_dev_struct 数据结构包括一个请求例程的地址和一个指针，指向一个 request 数据结构的列表，每一个都表达 buffer Cache 向设备读/写一块数据的一个请求。参见 drivers/block/ll_rw_blk.c 和 include/linux/blkdev.h。

当 buffer Cache 从一个已登记的设备读/写一块数据，或希望读/写一块数据到其他位置时，就在 blk_dev_struc 中增加一个 request 数据结构。每个 request 结构都有一个指向

华清远见 HQYJ.COM 华清远见教育集团官网：www.hqyj.com

一个或多个 buffer_head 数据结构的指针，每一个 request 结构都是读/写一块数据的请求。如果 buffer_head 数据结构被锁定（buffer Cache），可能会有一个进程在等待这个缓冲区的阻塞进程完成。每一个 request 结构都是从 all_request 表中分配的。如果 request 增加到空的 request 列表，就调用驱动程序的 request 函数处理这个 request 队列，否则，驱动程序只是简单地处理 request 队列中的每一个请求。

一旦设备驱动程序完成了一个请求，它就必须把每一个 buffer_head 结构从 request 结构中删除，标记它们为最新的，然后解锁。对于 buffer_head 的解锁会唤醒任何正在等待这个阻塞操作完成的进程。这样的例子包括文件解析的时候：必须等待 Ext2 文件系统从包括这个文件系统的块设备上读取包括下一个 Ext2 目录条目的数据块，这个进程会在将要包括目录条目的 buffer_head 队列中睡眠，直到设备驱动程序唤醒它。这个 request 数据结构会被标记为空闲，可以被另一个块请求使用。

字符设备和块设备的主要区别是：在对字符设备发出读/写请求时，实际的硬件 I/O 一般就紧接着发生了；块设备则不然，它利用一块系统内存作为缓冲区，当用户进程对设备请求能满足用户的要求时，就返回请求的数据，如果不能，就调用请求函数来进行实际的 I/O 操作。块设备主要是针对磁盘等慢速设备设计的，以免耗费过多的 CPU 时间来等待。

9.3.4 Linux 网络接口

为了屏蔽网络环境中物理网络设备的多样性，Linux 对所有的物理设备进行抽象并定义了一个统一的概念，称为接口（Interface）。所有对网络硬件的访问都是通过接口进行的，接口对上层协议提供一致化的操作集合来处理基本数据的发送和接收，对下层屏蔽硬件差异。

在 Linux 中所有网络接口都用一个 net_device 的数据结构表示。通常，网络设备是一个物理设备，如以太网卡，但软件也可以作为网络设备，如回环设备（Loopback）。所有被网络设备发送和接收的包都用数据结构 skb_buff 表示，这是一个具有很好灵活性的数据结构，可以很容易增加或删除网络协议数据包头。

9.3.5 Linux 设备文件

Linux 把所有外部设备按其数据交换的特性分为 3 类，无论哪个类型的设备，Linux 都把它统一当做文件来处理，可以像使用文件一样来使用这些设备。

- 字符设备是以字符为单位进行输入/输出的设备，如打印机、显示终端等。
- 块设备是以数据块为单位进行输入/输出的设备，如磁盘、光盘等。
- 网络设备是以数据包为单位进行数据交换的设备，如以太网卡等。

把设备看成文件具有以下几个含义：

- 每个设备具有一个文件名称，应用程序可以通过设备的文件名来访问具体的设

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

备，同时要受到文件系统访问权限控制机制的保护。

- 设备在内核中应该对应有一个索引节点。
- 设备应该可以以文件的方式进行操作。

把设备纳入文件管理体系以后，从用户的角度看，整个设备管理的层次结构如图 9.4 所示。

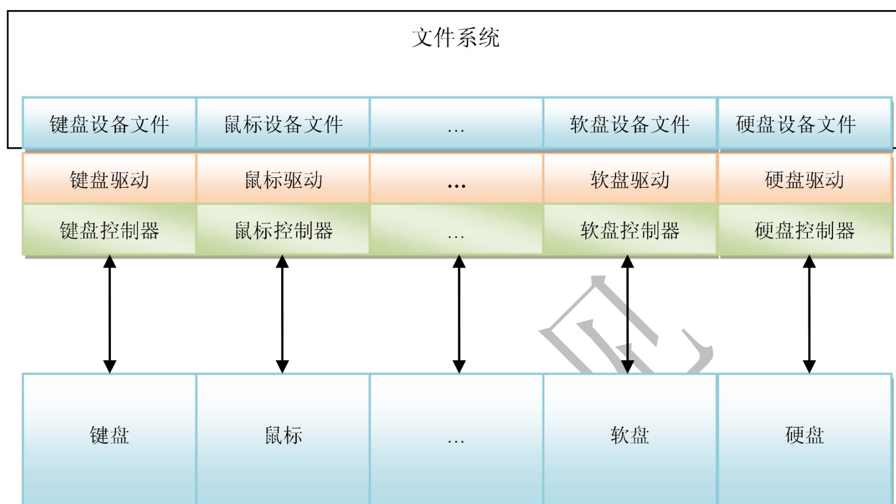


图 9.4　设备文件、驱动程序与外设的关系

设备以文件的形式出现在目录/dev 中，部分设备可读可写。文件列表能够看到设备的一些重要细节，如查看 ttyS0 的信息，代码如下：

```
ls -l /dev/ttyS0
crwxr-xr-x 1 root tty 4, 67 Apr 6 11:59 ttyS0
```

最左边的“c”表明这是一个字符设备。如果是“b”，则意味着“块设备”，“p”是先入先出设备（FIFO），“u”是非缓冲字符设备，“d”是目录，“l”是符号链接。数字“4”表示设备的主设备号，“67”是设备的次设备号。

那么设备编号如何获得？Linux 提供了专门的分配设备编号用的函数 register_chrdev_region() 和注销设备用的函数 unregister_chrdev()，它们都在 linux/fs/char_dev.c 实现。其代码如下：

```
int register_chrdev_region(dev_t from, unsigned count, const char *name)
{
    struct char_device_struct *cd;
    dev_t to = from + count;
    dev_t n, next;

    for (n = from; n < to; n = next) {
        next = MKDEV(MAJOR(n)+1, 0);
```


华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com




```
            if (next > to)
                next = to;
            cd = __register_chrdev_region(MAJOR(n), MINOR(n),
                    next - n, name);
            if (IS_ERR(cd))
                goto fail;
        }
        return 0;
    fail:
        to = n;
        for (n = from; n < to; n = next) {
            next = MKDEV(MAJOR(n)+1, 0);
            kfree(__unregister_chrdev_region(MAJOR(n), MINOR(n), next - n));
        }
        return PTR_ERR(cd);
    }

    void unregister_chrdev_region(dev_t from, unsigned count)
    {
        dev_t to = from + count;
        dev_t n, next;

        for (n = from; n < to; n = next) {
            next = MKDEV(MAJOR(n)+1, 0);
            if (next > to)
                next = to;
            kfree(__unregister_chrdev_region(MAJOR(n), MINOR(n), next - n));
        }
    }
```

9.3.6 Linux 设备注册与注销

在 Linux 系统中，完成驱动程序的第一个任务就是对设备进行注册，使用的函数是 device_register，这个函数在 drivers/base/core.c 中实现，代码如下：

```
    int device_register(struct device *dev)
    {
    device_initialize(dev);
        return device_add(dev);
    }
```

在 device_register 函数中，驱动核心初始化 device 结构体中的许多成员，向 kobject 核心注册设备的 kobject（导致热插拔事件产生），接着添加设备到其父节点所拥有的设备链表中。此后，所有的设备都可通过正确的顺序被访问，并知道其位于设备层次中的哪一点。

设备接着被添加到总线相关的设备链表（包含所有向总线注册的设备）中。接着驱动核心遍历这个链表，为每个驱动程序调用该总线的匹配函数。

在驱动程序中对设备进行注销的核心函数如下：

```
    void device_unregister(struct device *dev)
```


华清远见 HQYJ.COM 华清远见教育集团官网：www.hqyj.com

```
    {
        pr_debug("device: '%s': %s\n", dev->bus_id, __func__);
        device_del(dev);
        put_device(dev);
    }
```

在 device_unregister 函数中，驱动核心将删除这个设备的驱动程序指向这个设备的符号链接，并从它的内部设备链表中删除该设备，再以 device 结构中的 struct kobject 指针为参数，调用 kobject_del。kobject_del 函数引起用户空间的 hotplug 调用，表明 kobject 现在从系统中删除，接着删除所有该 kobject 以前创建的、与之相关联的 sysfs 文件和目录。kobject_del 函数也去除设备自身的 kobject 引用。此后，所有的和这个设备关联的 sysfs 入口被去除，并且和这个设备关联的内存被释放。

9.3.7 操作 I/O 端口

设备驱动程序是内核代码中与设备无关的软件和系统硬件之间的一个隔离带。因此，驱动程序在输入、输出和控制硬件设备的同时，必须要得到底层内核函数的支持，这些函数包括分配内存、申请中断、管理缓冲区等，由于驱动程序最终要通过 I/O 地址空间执行 I/O 端口操作，所以涉及硬件的操作就是内核提供的 I/O 端口操作函数。

在 PC 使用的系统总线中，有 8 位端口、16 位端口及 32 位端口。我们要注意，大部分与 I/O 端口操作相关的源代码都是和硬件平台相关的。Linux 内核头文件中(在与体系结构相关的头文件中)定义了如下一些内联函数。例如 S3C2410 的 I/O 端口操作定义（见 arch/arm/mach-s3c2410/include/mach/io.h）：

```
#define inb(p)  (__builtin_constant_p((p)) ? __inbc(p)    : __inb(p))
#define inw(p)  (__builtin_constant_p((p)) ? __inwc(p)    : __inw(p))
#define inl(p)  (__builtin_constant_p((p)) ? __inlc(p)    : __inl(p))
#define outb(v,p)(__builtin_constant_p((p)) ? __outbc(v,p) : __outb(v,p))
#define outw(v,p)(__builtin_constant_p((p)) ? __outwc(v,p) : __outw(v,p))
#define outl(v,p)(__builtin_constant_p((p)) ? __outlc(v,p) : __outl(v,p))
#define __ioaddr(p)   (__builtin_constant_p((p)) ? __ioaddr(p)  : __ioaddrc(p))
```

inb 函数是按字节（8 位宽度）读/写端口。port 参数在一些平台上定义为 unsigned long，而在另一些平台上定义为 unsigned short。不同平台上 inb 返回值的类型也不相同。内建函数 __builtin_constant_p 用于判断一个值是否为编译时常数，如果参数 p 的值是常数，函数返回 1，否则返回 0。此内建函数是 ARM 编译器支持的 GNU 编译器扩展。

inw/outw 用于访问 16 位端口；inl/outl 用于访问 32 位端口。这里没有定义 64 位的 I/O 操作。即使在 64 位的体系结构上，I/O 端口也只使用 32 位的数据通路。

当处理器和总线间数据传输过快时，一些平台（特别是在 X86）上的 I/O 操作可能会带来问题。问题源于相对 ISA 总线处理器的时钟频率太快了，当设备卡太慢时，这个问题就容易暴露出来。解决该问题的方法是，如果后面又跟着一条 I/O 指令，就在该条 I/O 指令后添加一段延迟。如果设备会丢失数据，或者担心它会丢失数据，可以用暂停

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

式的 I/O 操作取代通常的 I/O 操作。暂停式 I/O 函数类似前面列出的那些 I/O 函数，但它们的名称都以_p 结尾，如 inb_p、outb_p 等。对所有支持的体系结构，如果定义了不暂停的 I/O 函数，那么也会定义相应的暂停式 I/O 函数，虽然有些平台上它们会被扩展成相同的代码，例如：

```
#define outb_p(val,port)  outb((val),(port))
#define outw_p(val,port)  outw((val),(port))
#define outl_p(val,port)  outl((val),(port))
#define inb_p(port)       inb((port))
#define inw_p(port)       inw((port))
#define inl_p(port)       inl((port))

#define outsb_p(port,from,len) outsb(port,from,len)
#define outsw_p(port,from,len) outsw(port,from,len)
#define outsl_p(port,from,len) outsl(port,from,len)
#define insb_p(port,to,len)    insb(port,to,len)
#define insw_p(port,to,len)    insw(port,to,len)
#define insl_p(port,to,len)    insl(port,to,len)
```

新的 I/O 内存接口由下面这些函数组成：

```
#define readb(c)          (__readwrite_bug("readb"),0)
#define readw(c)          (__readwrite_bug("readw"),0)
#define readl(c)          (__readwrite_bug("readl"),0)
#define writeb(v,c)       __readwrite_bug("writeb")
#define writew(v,c)       __readwrite_bug("writew")
#define writel(v,c)       __readwrite_bug("writel")
```

这些函数（宏）用于读/写 8 位、16 位和 32 位的数据项。

9.3.8 Linux 逻辑 I/O 与设备驱动程序的接口

Linux 内核为进程提供了使用驱动程序的接口。由于 Linux 把设备作为文件，所以访问设备驱动程序的接口与访问文件的接口是统一的，即 VFS。Linux 对设备的管理和控制是使用 VFS 提供的各种数据结构和操作函数实现的。

逻辑 I/O 层与设备驱动程序之间的接口是由两个由内核分别定义的数据结构组成的数组。

第 8 章中曾讲过，字符设备通过 file_operations 结构把驱动的操作和设备号联系在一起，每个被打开的文件都对应于一系列操作，用来执行一系列系统调用。块设备同样也拥有一个操作接口：block_device_operations，该接口定义了 open、close、ioctl 等函数接口。这个结构在 include\linux\blkdev.h 中定义，代码如下：

```
struct block_device_operations {
     int (*open) (struct block_device *, fmode_t);
     int (*release) (struct gendisk *, fmode_t);
     int (*locked_ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
     int (*ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
     int (*compat_ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
```


华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

```
        int (*direct_access) (struct block_device *, sector_t,   void **, unsigned
long *);
        int (*media_changed) (struct gendisk *);
        int (*revalidate_disk) (struct gendisk *);
        int (*getgeo)(struct block_device *, struct hd_geometry *);
        struct module *owner;
    };
```

读者可能发现，在这个结构中并没有 read、write 接口。我们知道，字符设备的实现比较简单，内核例程和用户态 API 一一对应，用户层的 read 函数直接对应了内核中的 read 例程，这种映射关系由字符设备的 file_operations 维护。编写块设备驱动程序更复杂，同时对效率也有很高的要求。为了高效进行大数据量的传输，Linux 内核没有像字符设备那样提供 read、write 接口，而是直接到文件系统层，然后再由文件系统层发起读/写请求。

在 Linux 内核中，使用 gendisk（通用磁盘）结构体来表示一个独立的磁盘设备（或分区），这个结构体的定义如下（见 include\linux\genhd.h）：

```
    struct gendisk {
        /* major, first_minor and minors are input parameters only,
         * don't use directly.  Use disk_devt() and disk_max_parts().
         */
        int major;             /* major number of driver */
        int first_minor;
        int minors;
    /* maximum number of minors, =1 for disks that can't be partitioned. */

        char disk_name[DISK_NAME_LEN]; /* name of major driver */
        struct disk_part_tbl *part_tbl;
        struct hd_struct part0;

        struct block_device_operations *fops;
        struct request_queue *queue;
        void *private_data;

        int flags;
        struct device *driverfs_dev;  // FIXME: remove
        struct kobject *slave_dir;

        struct timer_rand_state *random;

        atomic_t sync_io;      /* RAID */
        struct work_struct async_notify;
    #ifdef  CONFIG_BLK_DEV_INTEGRITY
        struct blk_integrity *integrity;
    #endif
        int node_id;
    };
```

major、first_minor 和 minors 共同表征了磁盘的主、次设备号，同一个磁盘的各个分区共享一个主设备号，而从设备号则不同。fops 为 block_device_operations，即块设备

操作集合。queue 是内核用来管理这个设备的 I/O 请求队列的指针。capacity 表明设备的容量，以 512B 为单位。private_data 可用于指向磁盘的任何私有数据，用法与字符设备驱动 file 结构体的 private_data 类似。Linux 内核提供了一组函数来操作 gendisk，主要包括分配、增加、释放和设置容量等。

在 Linux 块设备驱动中，使用 request 结构体来表征等待进行的 I/O 请求。这个结构体的定义如下（见 include\linux\blkdev.h）：

```
struct request {
    struct list_head queuelist;
    struct call_single_data csd;
    int cpu;
    struct request_queue *q;
    unsigned int cmd_flags;
    enum rq_cmd_type_bits cmd_type;
    unsigned long atomic_flags;
    sector_t sector;        /* next sector to submit */
    sector_t hard_sector;       /* next sector to complete */
    unsigned long nr_sectors; /* no. of sectors left to submit */
    unsigned long hard_nr_sectors; /* no. of sectors left to complete */
    /* no. of sectors left to submit in the current segment */
    unsigned int current_nr_sectors;
    /* no. of sectors left to complete in the current segment */
    unsigned int hard_cur_sectors;
    struct bio *bio;
    struct bio *biotail;
    struct hlist_node hash;    /* merge hash */
    union {
        struct rb_node rb_node;    /* sort/lookup */
        void *completion_data;
    };
    void *elevator_private;
    void *elevator_private2;

    struct gendisk *rq_disk;
    unsigned long start_time;
    unsigned short nr_phys_segments;
    unsigned short ioprio;
    void *special;
    char *buffer;
    int tag;
    int errors;
    int ref_count;
    unsigned short cmd_len;
    unsigned char __cmd[BLK_MAX_CDB];
    unsigned char *cmd;
    unsigned int data_len;
    unsigned int extra_len;    /* length of alignment and padding */
    unsigned int sense_len;
    void *data;
    void *sense;
```


华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

```
        unsigned long deadline;
        struct list_head timeout_list;
        unsigned int timeout;
        int retries;
        rq_end_io_fn *end_io;
        void *end_io_data;
        /* for bidi */
        struct request *next_rq;
    };
```

hard_sector、hard_nr_sectors、hard_cur_sectors 这 3 个成员标识还未完成的扇区，hard_sector 是第一个尚未传输的扇区，hard_nr_sectors 是待完成的扇区数，hard_cur_sectors 是当前 I/O 操作中待完成的扇区数。这些成员只用于内核块设备层，驱动不应当使用它们。在驱动程序中经常使用的是 sector、nr_sectors、current_nr_sectors。这 3 个成员在内核和驱动交互中发挥着重大作用。它们以 512B 大小为一个扇区，如果硬件的扇区大小不是 512B，则需要进行相应的调整。例如，如果硬件的扇区大小是 2048B，则在进行硬件操作之前，需要用 4 来除起始扇区号。

bio 是这个请求中包含的 bio 结构体的链表，buffer 是指向缓冲区的指针，数据应当被传送到或来自这个缓冲区，这个指针是一个内核虚拟地址，可被驱动直接引用。

块设备驱动程序的每个请求描述了一组连续的物理扇区与一组请求块之间的对应关系。当块设备驱动程序每读/写完成一个块时，就通知块缓冲标记此块的状态，同时为读/写下一块的状态做好准备。下面要学习的就是请求队列。它的实现如下：

```
    struct request_queue
    {
        struct list_head  queue_head;
        struct request         *last_merge;
        elevator_t             *elevator;
        struct request_list    rq;

        request_fn_proc        *request_fn;
        make_request_fn        *make_request_fn;
        prep_rq_fn             *prep_rq_fn;
        unplug_fn              *unplug_fn;
        prepare_discard_fn     *prepare_discard_fn;
        merge_bvec_fn      *merge_bvec_fn;
        prepare_flush_fn  *prepare_flush_fn;
        softirq_done_fn        *softirq_done_fn;
        rq_timed_out_fn        *rq_timed_out_fn;
        dma_drain_needed_fn    *dma_drain_needed;
        lld_busy_fn        *lld_busy_fn;
        sector_t       end_sector;
        struct request         *boundary_rq;

        struct timer_list unplug_timer;
        int            unplug_thresh;    /* After this many requests */
        unsigned long      unplug_delay;/* After this many jiffies */
        struct work_struct     unplug_work;
```


华清远见 HQYJ.COM

华清远见教育集团官网：www.hqyj.com

```
    struct backing_dev_info    backing_dev_info;
    void          *queuedata;
    unsigned long     bounce_pfn;
    gfp_t             bounce_gfp;
    unsigned long     queue_flags;
    spinlock_t        __queue_lock;
    spinlock_t        *queue_lock;
    struct kobject kobj;
    unsigned long     nr_requests; /* Max # of requests */
    unsigned int      nr_congestion_on;
    unsigned int      nr_congestion_off;
    unsigned int      nr_batching;

    unsigned int      max_sectors;
    unsigned int      max_hw_sectors;
    unsigned short        max_phys_segments;
    unsigned short        max_hw_segments;
    unsigned short        hardsect_size;
    unsigned int      max_segment_size;

    unsigned long     seg_boundary_mask;
    void              *dma_drain_buffer;
    unsigned int      dma_drain_size;
    unsigned int      dma_pad_mask;
    unsigned int      dma_alignment;

    struct blk_queue_tag  *queue_tags;
    struct list_head  tag_busy_list;

    unsigned int      nr_sorted;
    unsigned int      in_flight;

    unsigned int      rq_timeout;
    struct timer_list timeout;
    struct list_head  timeout_list;
    unsigned int      sg_timeout;
    unsigned int      sg_reserved_size;
    int           node;
#ifdef CONFIG_BLK_DEV_IO_TRACE
    struct blk_trace  *blk_trace;
#endif
    unsigned int      ordered, next_ordered, ordseq;
    int           orderr, ordcolor;
    struct request        pre_flush_rq, bar_rq, post_flush_rq;
    struct request        *orig_bar_rq;

    struct mutex      sysfs_lock;

#if defined(CONFIG_BLK_DEV_BSG)
    struct bsg_class_device bsg_dev;
#endif
    struct blk_cmd_filter cmd_filter;
};
```

请求队列跟踪等候的块 I/O 请求，它存储用于描述这个设备能够支持的请求的类型信息、它们的最大大小、多少不同的段可进入一个请求、硬件扇区大小、对齐要求等参数，其结果是：如果请求队列被配置正确了，它不会交给该设备一个不能处理的请求。请求队列还实现一个插入接口，这个接口允许使用多个 I/O 调度器，I/O 调度器的工作是以最优性能的方式向驱动提交 I/O 请求。大部分 I/O 调度器累积批量的 I/O 请求，并将它们排列为递增（或递减）的块索引顺序后提交给驱动。进行这些工作的原因在于，对于磁头而言，当给定顺序排列的请求时，可以使得磁盘顺序地从一头到另一头工作，非常像一个满载的电梯，在一个方向移动直到所有它的“请求”已被满足。另外，I/O 调度器还负责合并邻近的请求，当一个新 I/O 请求被提交给调度器后，它会在队列中搜寻包含邻近扇区的请求；如果找到一个，并且结果的请求不是太大，调度器将合并这两个请求。

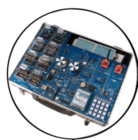
由以上内容可知，块设备驱动编程的主要工作包括分配并初始化一个 gendisk 结构、分配并初始化一个请求队列、请求处理函数的编写（request_fn）和中断的处理等。

9.4 本章习题



1．简单说明设备的分类。
2．I/O 空间编址有几种方式？分别是什么？
3．Linux 系统的设备驱动一般分几类？各有什么特点？
4．什么是设备节点？如何创建？
5．什么是设备号？在驱动程序中如何申请？

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com



华清远见
HQYJ.COM

第10章 嵌入式Linux的构建

构建开发环境是任何开发工作的基础，对于软、硬件非常丰富的嵌入式系统来说，构建高效、稳定的环境是能否开展工作的重要因素之一。本章将介绍如何构建一套嵌入式Linux开发环境。在构建开发环境之前，有必要了解一下嵌入式Linux的开发流程。因为嵌入式Linux开发往往会涉及多个层面，这与桌面开发有很大的不同。

华清远见

华清远见
HQYJ.COM
华清远见教育集团官网：www.hqyj.com

10.1 嵌入式开发环境的搭建



10.1.1 嵌入式交叉编译环境的搭建

搭建交叉编译环境是嵌入式开发的第一步，也是必备的一步。搭建交叉编译环境的方法很多，不同的体系结构、不同的操作内容甚至是不同版本的内核，都会用到不同的交叉编译器，而且，有些交叉编译器经常会有部分 BUG，这都会导致最后的代码无法正常运行。因此，选择合适的交叉编译器对于嵌入式开发是非常重要的。

交叉编译器完整的安装一般涉及多个软件的安装（读者可以从 ftp://gcc.gnu.org/pub/下载），包括 binutils、gcc、glibc 等软件。其中，binutils 主要用于生成一些辅助工具，如 objdump、as、ld 等；gcc 是用来生成交叉编译器，主要生成 arm-linux-gcc 交叉编译工具（应该说，生成此工具后已经搭建起了交叉编译环境，可以编译 Linux 内核了，但由于没有提供标准用户函数库，用户程序还无法编译）；glibc 主要是提供用户程序所使用的一些基本的函数库。这样，交叉编译环境就完全搭建起来了。

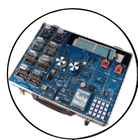
上面所述的搭建交叉编译环境比较复杂，很多步骤都涉及对硬件平台的选择。因此，现在提供开发板的公司一般会在附赠的光盘中提供该公司测试通过的交叉编译器，而且很多公司把以上安装步骤全部写入脚本文件或以发行包的形式提供，这样就大大方便了用户的使用。如优龙公司的开发光盘中就附带了 2.95.3 和 3.3.2 两个版本的交叉编译器，其中，前一个版本是用于编译 Linux 2.4 内核的，后一个版本是用于编译 Linux 2.6 版本内核的。由于这是厂商测试通过的编译器，因此可靠性比较高，而且与开发板能够很好地吻合。所以推荐初学者直接使用厂商提供的编译器。当然，由于时间滞后的原因，这个编译器往往不是最新版本的，若需要更新，读者可另外查找相关资料学习。本书就以优龙自带的 cross-3.3.2 为例进行讲解（具体的名称不同厂商可能会有区别）。

在/usr/local/arm 下解压 cross-3.3.2.bar.bz2，代码如下：

```
[root@localhost arm]# tar -jxvf cross-3.3.2.bar.bz2
[root@localhost arm]# ls
3.3.2  cross-3.3.2.tar.bz2
[root@localhost arm]# cd ./3.3.2
[root@localhost arm]# ls
arm-linux  bin  etc  include  info  lib  libexec  man  sbin  share  VERSIONS
[root@localhost bin]# which arm-linux*
/usr/local/arm/3.3.2/bin/arm-linux-addr2line
/usr/local/arm/3.3.2/bin/arm-linux-ar
/usr/local/arm/3.3.2/bin/arm-linux-as
/usr/local/arm/3.3.2/bin/arm-linux-c++
/usr/local/arm/3.3.2/bin/arm-linux-c++filt
/usr/local/arm/3.3.2/bin/arm-linux-cpp
```

华清远见

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com




```
/usr/local/arm/3.3.2/bin/arm-linux-g++
/usr/local/arm/3.3.2/bin/arm-linux-gcc
/usr/local/arm/3.3.2/bin/arm-linux-gcc-3.3.2
/usr/local/arm/3.3.2/bin/arm-linux-gccbug
/usr/local/arm/3.3.2/bin/arm-linux-gcov
/usr/local/arm/3.3.2/bin/arm-linux-ld
/usr/local/arm/3.3.2/bin/arm-linux-nm
/usr/local/arm/3.3.2/bin/arm-linux-objcopy
/usr/local/arm/3.3.2/bin/arm-linux-objdump
/usr/local/arm/3.3.2/bin/arm-linux-ranlib
/usr/local/arm/3.3.2/bin/arm-linux-readelf
/usr/local/arm/3.3.2/bin/arm-linux-size
/usr/local/arm/3.3.2/bin/arm-linux-strings
/usr/local/arm/3.3.2/bin/arm-linux-strip
```

可以看到，在/usr/local/arm/3.3.2/bin/文件是下已经安装了很多交叉编译工具。用户可以查看 arm 文件夹下的 Versions 文件，显示如下：

```
Versions
  gcc-3.3.2
  glibc-2.3.2
  binutils-head
Tool chain binutils configuration:
../binutils-head/configure …
Tool chain glibc configuration:
../glibc-2.3.2/configure …
Tool chain gcc configuration
../gcc-3.3.2/configure …
```

可以看到，优龙公司提供的交叉编译工具确实集成了 binutils、gcc、glibc 这几个软件，每个软件也都有比较复杂的配置信息，读者可以查看 Versions 文件了解相关信息。

10.1.2 超级终端和 Minicom 配置及使用

前文已知，嵌入式系统开发的程序运行环境是在硬件开发板上，那么如何把开发板上的信息显示给开发人员呢？最常用的就是通过串口线输出到宿主机的显示器上，这样，开发人员就可以看到系统的运行情况了。在 Windows 和 Linux 系统中都有不少串口通信软件，可以很方便地对串口进行配置，其中，最主要的配置参数就是波特率、数据位、停止位、奇偶校验位和数据流控制位等，但是它们一定要根据实际情况进行相应配置。下面介绍 Windows 系统中典型的串口通信软件“超级终端”和在 Linux 系统下的“Minicom”。

1. 超级终端

首先，在 Windows 系统下选择“开始”→“附件”→“通讯”→“超级终端”命令，会打开如图 10.1 所示的新建超级终端界面，在“名称”文本框中可随意输入该连接的名称。单击“确定”开始对新建超级终端进行设置，如图 10.2 所示。

接下来在图 10.2 中将“连接时使用”的方式改为“COM1”，即通过串口 1。

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

接下来就到了最关键的一步——设置串口连接参数。注意，每块开发板的连接参数可能会有差异，其中的具体数据在开发商提供的用户手册中会有说明。如优龙公司的这款 FS2410 采用的是波特率为 115200，数据为 8 位，无奇偶校验位，停止位 1，无硬件流，其对应配置如图 10.3 所示。

这样，就基本完成了配置，最后单击“确定”按钮就可以了。这时，读者可以把开发板的串口线和 PC 相连，若配置正确，在开发板上电后在超级终端的窗口中应能显示如图 10.4 所示的串口信息。

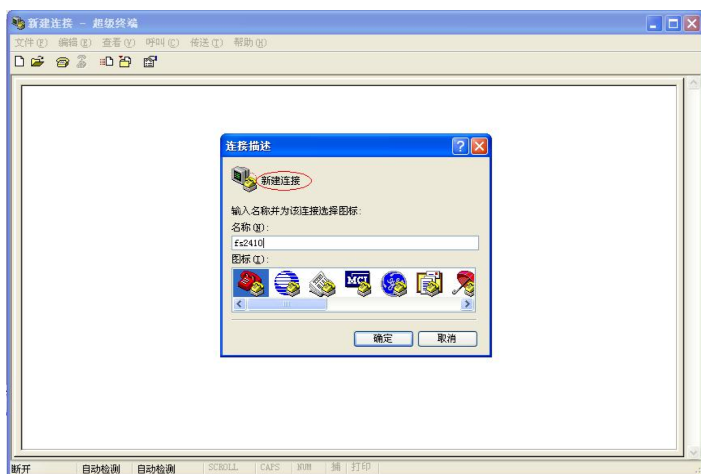


图 10.1　新建超级终端界面

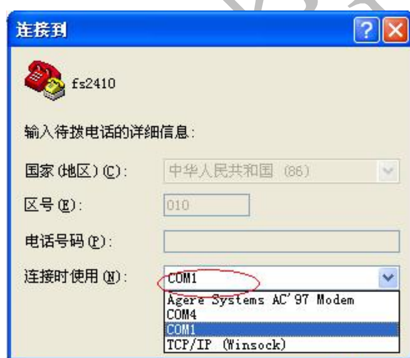


图 10.2　选择连接时使用方式



图 10.3　配置串口相关参数

2．Minicom

Minicom 是 Linux 系统下串口通信的软件，它的使用完全依靠键盘的操作，虽然没有“超级终端”那么易用，但是使用习惯之后读者将会体会到它的高效与便利。下面主要讲解如何对 Minicom 进行串口参数的配置。

华清远见
HQYJ.COM
华清远见教育集团官网：www.hqyj.com



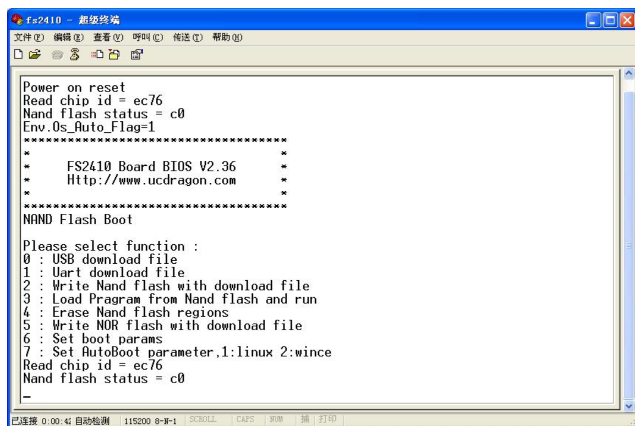


图 10.4 串口相关信息

首先，在命令行中输入“minicom”命令，这样就启动了 Minicom 软件（如果系统没有安装 Minicom，可以运行“sudo apt-get install Minicom”进行安装）。Minicom 在启动时默认会进行初始化配置，如图 10.5 所示。

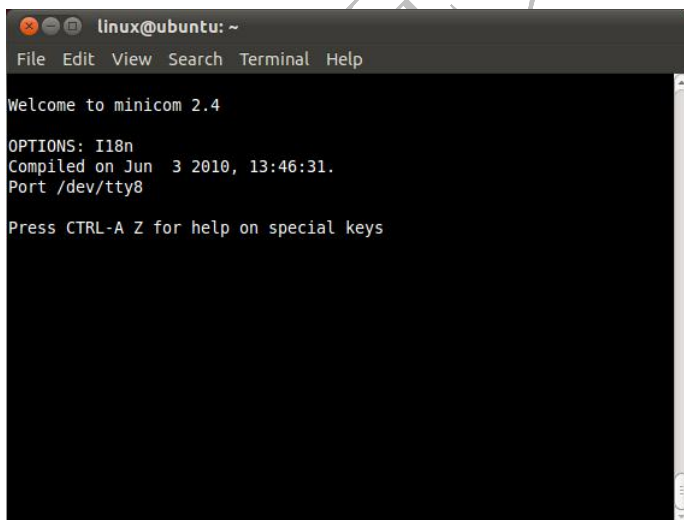


图 10.5 Minicom 启动

按照图 10.5 中的提示，按“Ctrl+A+Z”组合键，来查看 minicom 的帮助，如图 10.6 所示。

按照帮助所示，可按“O”键（代表 Configure Minicom）来配置 Minicom 的串口参数，当然也可以直接按“Ctrl+A+O” 组合键来进行配置，如图 10.7 所示。

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

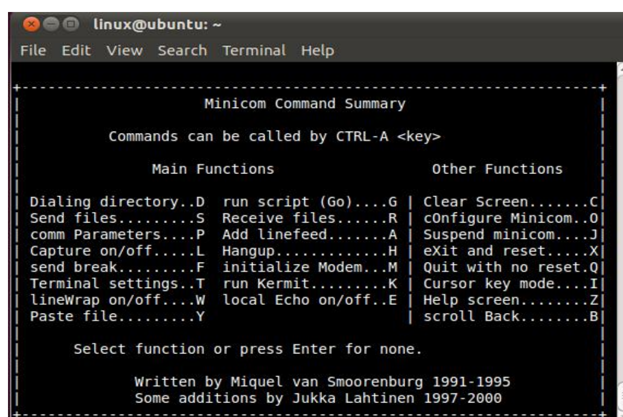


图 10.6　Minicom 帮助

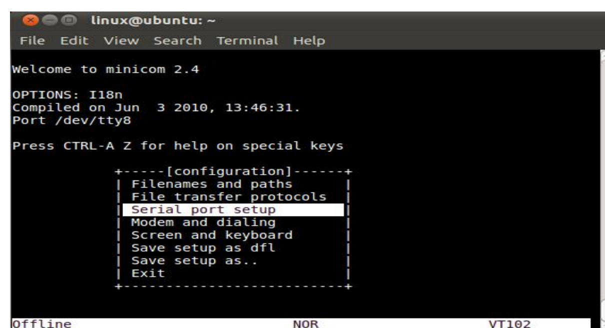


图 10.7　Minicom 配置界面

在这个配置框中选择“Serial port setup”子项，进入如图 10.8 所示的配置界面。

该界面中列出的配置是 Minicom 启动时的默认配置，用户可以通过输入每一项前的大写字母，分别对每一项进行更改。如图 10.9 所示为在“Change which setting 中”输入了“A”，此时光标转移到第 A 项的对应处。

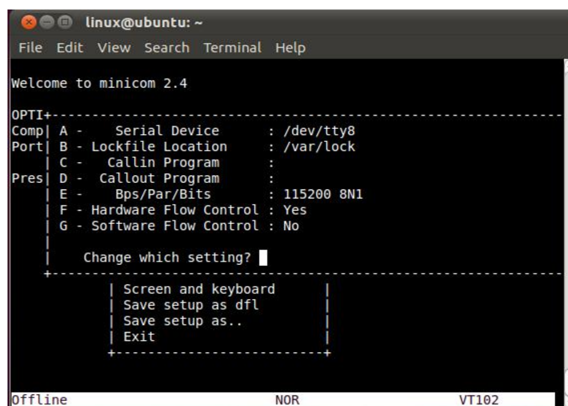


图 10.8　Minicom 串口属性配置界面

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

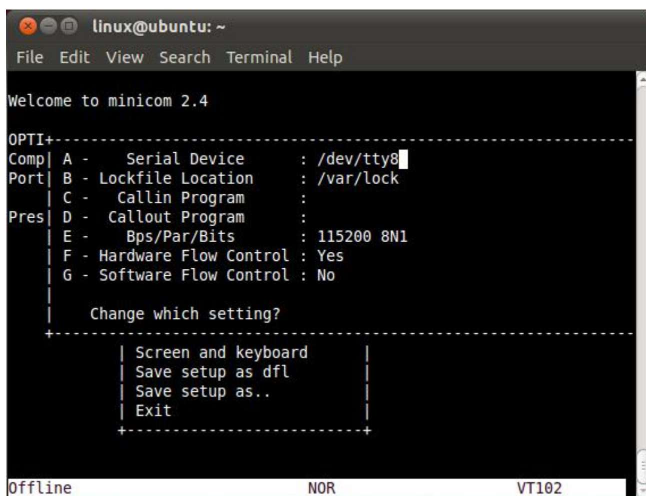


图 10.9 Minicom 串口号配置

接下来，要对波特率、数据位和停止位进行配置，按“E”键，进入如图 10.10 所示的配置界面。

在该配置界面中输入相应波特率、停止位等对应的字母，即可实现配置，配置完成后按回车键就退出了该配置界面，在上层界面中显示如图 10.11 所示的配置信息，注意与图 10.8 进行对比，确定相应参数是否已被重新配置。

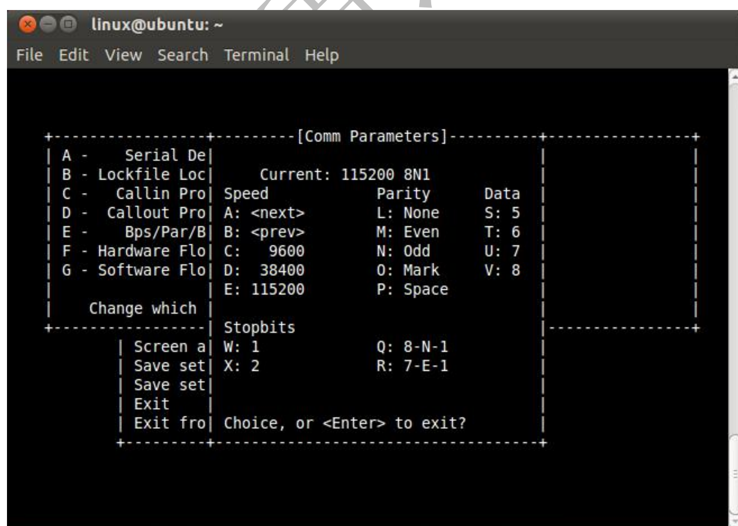


图 10.10 Minicom 波特率等配置界面

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

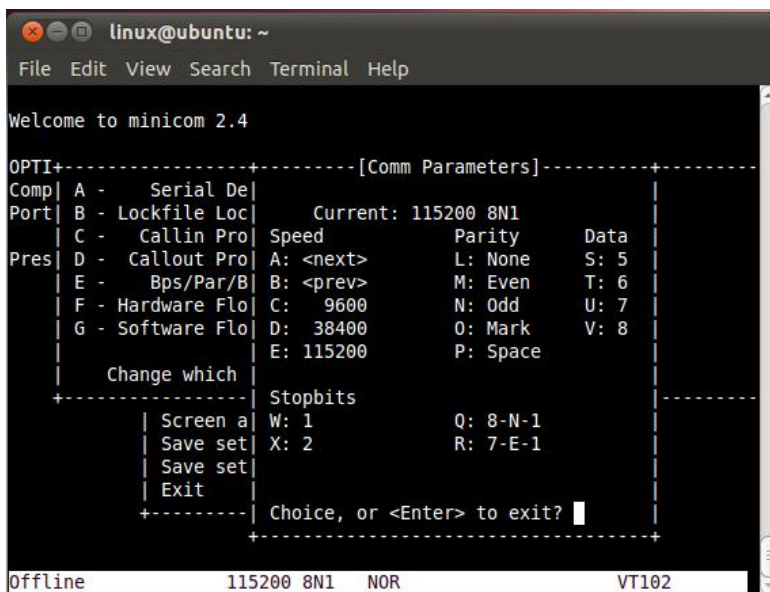


图 10.11 Minicom 配置完成后界面

确认配置正确后，可按回车键返回上一级配置界面，并将其保存为默认配置，如图 10.12 所示。

之后，可重新启动 Minicom 使刚才的配置生效，连上开发板的串口线之后，就可在 Minicom 中打印出正确的串口信息，如图 10.13 所示。

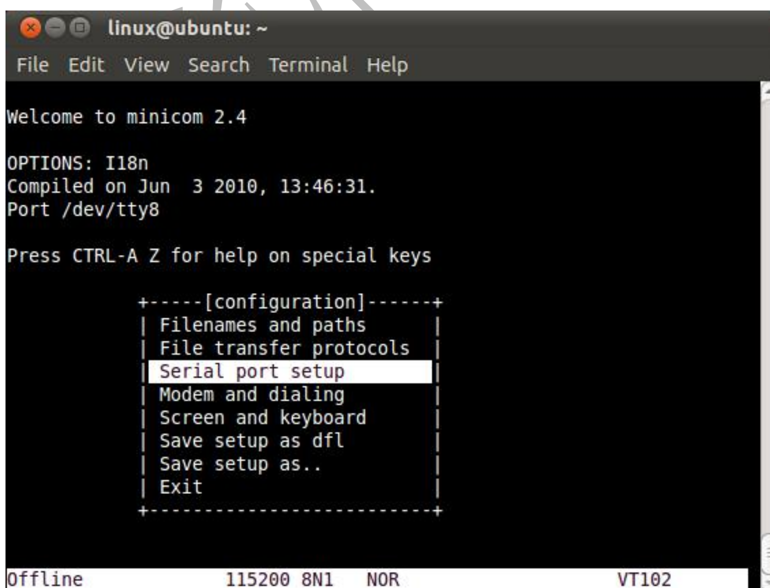


图 10.12 Minicom 保存配置信息

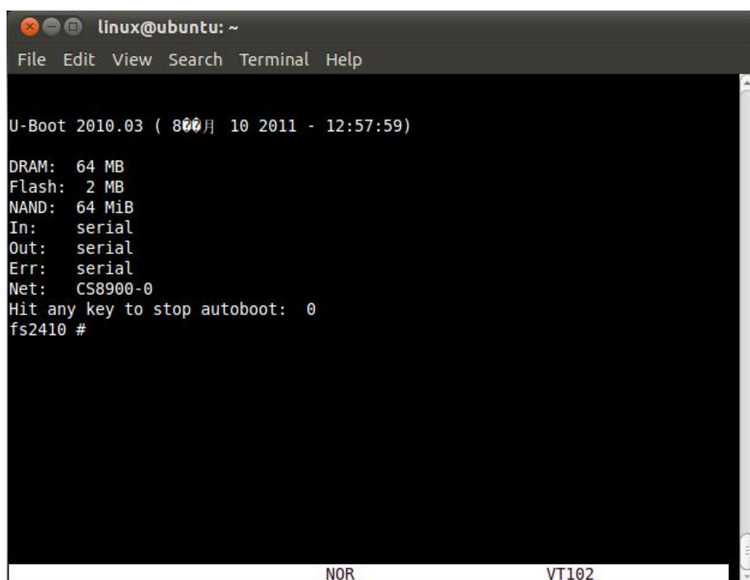


图 10.13　Minicom 显示串口信息

至此，读者应该能将开发板的系统情况通过串口打印到宿主机上了，这样，就能很好地了解硬件的运行状况。

10.1.3　下载映像（Image）到开发板

嵌入式开发的运行环境是目标板，而开发环境是宿主机。因此，需要把宿主机中经过编译之后的可执行文件下载到目标板上去。需要注意的是，这里所讲的下载是下载到目标机中的 SDRAM。然后，用户可以选择直接从 SDRAM 中运行或写入到 Flash 中再运行。运行常见的下载方式有网络下载（如 TFTP、FTP 等方式）、串口下载、USB 下载等，本书主要讲解网络下载中的 TFTP 方式和串口下载方式。

1．TFTP

TFTP 协议是简单文件传输协议，它可以看做是一个 FTP 协议的简化版本，与 FTP 协议相比，它的最大区别在于没有用户管理的功能。TFTP 的传输速度快，可以通过防火墙，使用方便快捷，因此在嵌入式的文件传输中广泛使用。

同 FTP 一样，TFTP 分为客户端和服务器端两种。通常，首先在宿主机上开启 TFTP 服务器端服务，设置好 TFTP 的根目录内容（也就是供客户端下载的文件），接着，在目标板上开启 TFTP 的客户端程序（现在很多开发板都已经提供了该项功能）。这样，把目标板和宿主机用直连线相连之后，就可以通过 TFTP 协议传输可执行文件了。

下面分别讲述在 Linux 和 Windows 系统下的配置方法。

华清远见

1）Linux 系统下 TFTP 服务配置

Linux 系统下 TFTP 的服务器服务是由 xinetd 所设定的，默认情况下处于关闭状态。首先，要修改 TFTP 的配置文件，开启 TFTP 服务，代码如下：

```
[root@ycw tftpboot]# vi /etc/xinetd.d/tftp
# default: off
# description: The tftp server serves files using the trivial file transfer \
#          protocol.  The tftp protocol is often used to boot diskless \
#          workstations, download configuration files to network-aware printers, \
#          and to start the installation process for some operating systems.
service tftp
{
              socket_type               = dgram
              protocol                  = udp
              wait                      = yes
              user                      = root
              server                    = /usr/sbin/in.tftpd
              server_args               = -s /tftpboot
              disable                   = no
              per_source                = 11
              cps                       = 100 2
              flags                     = IPv4
}
```

在这里，主要将“disable=yes”改为“no”；另外，从“server_args”可以看出，TFTP 服务器端的默认根目录为“/tftpboot”，用户若需要可以更改为其他目录。

然后，重启 xinetd 服务，使刚才的更改生效，代码如下：

```
[root@ycw tftpboot]# service xinetd restart
关闭 xinetd:                                          [  确定  ]
启动 xinetd:                                          [  确定  ]
```

使用命令“netstat -au”以确认 TFTP 服务是否已经开启，代码如下：

```
[root@ycw tftpboot]# netstat -au
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address             Foreign Address      State
udp        0      0 *:32768                   *:*
udp        0      0 *:831                     *:*
udp        0      0 *:tftp                    *:*
udp        0      0 *:sunrpc              *:*
udp        0      0 *:ipp                     *:*
```

这时，用户就可以把所需要的传输文件放到“/tftpboot”目录下，这样，主机上的 TFTP 服务就可以建立起来了。

最后，用直连线（注意：不可以使用网线）把目标板和宿主机连起来，并且将其配置成一个网段的地址，再在目标板上启动 TFTP 客户端程序（注意：不同的开发板所使用的命令可能会不同，读者可以查看“帮助”来获得确切的命令名及格式），代码如下：

```
=>tftpboot 0x30200000 zImage
TFTP from server 192.168.1.1; our IP address is 192.168.1.100
```


华清远见 HQYJ.COM 华清远见教育集团官网：www.hqyj.com

```
Filename 'zImage'.
Load address: 0x30200000
Loading: #################################################################
         ###############################################################
         #############################################
done
Bytes transferred = 881988 (d7544 hex)
```

可以看到，此处目标板使用的 IP 为“192.168.1.100”，宿主机使用的 IP 为“192.168.1.1”，下载到目标板的地址为 0x30200000，文件名为“zImage”。

2）Windows 系统下 TFTP 服务配置

在 Windows 系统下配置为 TFTP 服务器端需要下载 TFTP 服务器软件，常见的为 TFTPd32。

首先，单击 TFTPD32 下方的设置按钮，进入设置界面，如图 10.14 所示，在这里，主要配置 TFTP 服务器端地址，也就是本机的地址。

接下来，重新启动 TFTPD32 软件使刚才的配置生效，这样服务器端的配置就完成了，这时，就可以用直连线连接目标机和宿主机，且在目标机上开启 TFTP 服务进行文件传输，这时，TFTP 服务器端如图 10.15 和图 10.16 所示。

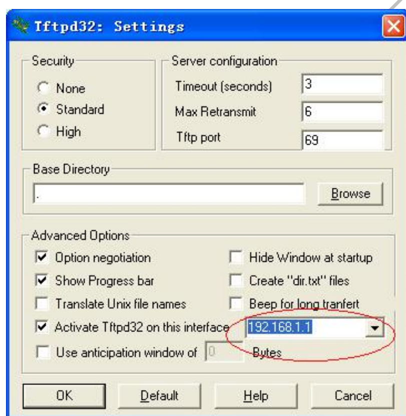


图 10.14　TFTPD32 配置界面

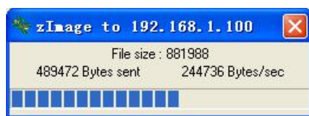


图 10.15　TFTP 文件传输

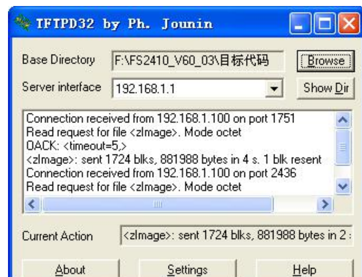


图 10.16　TFTP 服务器端显示情况

2. 串口下载

使用串口下载需要配合特定的下载软件，如优龙公司提供的 DNW 软件，一般在 Windows 下进行操作。虽然串口下载的速度没有网络下载快，但由于它很方便，不需要额外连线和设置 IP 等操作，因此也广受用户的青睐。下面就以 DNW 软件为例，介绍串口下载的方式。

与其他串口通信的软件一样，在 DNW 中也要设置“波特率”、“端口号”等。打开“Configuration”下的“Options”界面，如图 10.17 所示。

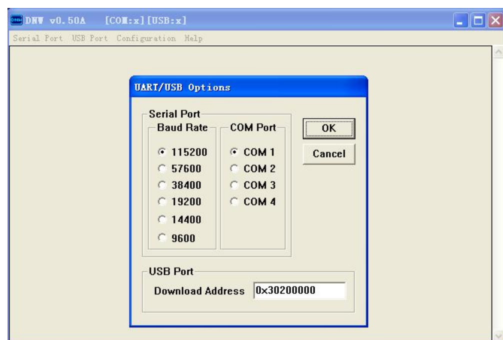


图 10.17　DNW 配置界面

在配置完之后，选择“Serial Port”→“Connect”命令，再将开发板上电，选择“串口下载”，接着选择“Serial Port”→“Transmit” 命令，这时，就可以进行文件传输了，如图 10.18 和图 10.19 所示。这里，DNW 默认串口下载的地址为 0x30200000。

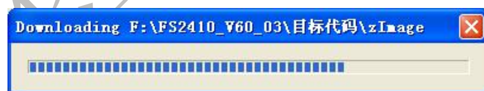


图 10.18　DNW 串口下载图

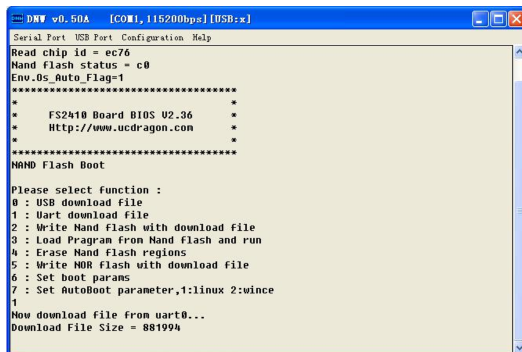


图 10.19　DNW 串口下载情形图

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

10.1.4　编译嵌入式 Linux 内核

做完了前期的准备工作之后，就可以编译嵌入式移植 Linux 的内核了。在这里，本书主要介绍嵌入式 Linux 内核的编译过程，后面会进一步介绍嵌入式 Linux 中体系结构相关的内核代码，读者在此之后就可以尝试嵌入式 Linux 操作系统的移植。

Linux 内核中不同的目录结构中都有相应的 Makefile，而不同的 Makefile 又通过彼此之间的依赖关系构成统一的整体，共同完成建立依存关系、建立内核等功能。

内核的编译根据不同的情况会有不同的步骤，但其中主要包括 3 个步骤：内核配置、建立依存关系、建立内核，其他的为一些辅助功能，如清除文件等。读者在实际编译时若出现错误，可以考虑采用其他辅助功能。下面分别讲述这 3 个主要的步骤。

1．内核配置

内核配置中的选项主要是用户用来为目标板选择处理器架构的选项，不同的处理器架构会有不同的处理器选项，如 ARM 就有其专用的选项，如“Multimedia capabilities port drivers”等。因此，在此之前，必须确保在根目录的 Makefile 中“ARCH”的值已设定了目标板的类型，如：

```
ARCH      := arm
```

接下来就可以进行内核配置了，内核支持 4 种不同的配置方法，这几种方法只是与用户交互的界面不同，其实现的功能是一样的。每种方法都会读入一个默认的配置文件——根目录下“.config”隐藏文件（用户也可以手动修改该文件，但不推荐使用）。当然，用户也可以自己加载其他配置文件，也可以将当前的配置保存为其他名称的配置文件。这 4 种方式如下。

- make config：基于文本的最为传统的配置界面，不推荐使用。
- make menuconfig：基于文本选单的配置界面，字符终端下推荐使用。
- make xconfig：基于图形窗口模式的配置界面，X-Window 下推荐使用。
- make oldconfig：自动读入“.config”配置文件，并且只要求用户设定上一次没有设定过的选项。

在这 4 种模式中，make menuconfig 使用最为广泛，下面就以 make menuconfig 为例进行讲解，如图 10.20 所示。

从图 10.20 中可以看出，Linux 内核允许用户对其各类功能逐项配置，一共有 18 类配置选项，这里就不对这 18 类配置选项进行一一讲解了，需要的读者可以参见相关选项的 help。在 menuconfig 的配置界面中是纯键盘的操作，用户可使用上下方向键和“Tab”键移动光标以进入相关子项，如图 10.21 所示为进入了“System Type”子项的界面，该子项是一个重要的选项，主要用来选择处理器的类型。

可以看到，每个选项前都有一个括号，通过按空格键或“Y”键表示包含该选项，按“N”键表示不包含该选项。

华清远见 HQYJ.COM　华清远见教育集团官网：www.hqyj.com

另外，读者可能注意到，这里的括号有 3 种，即中括号、尖括号或圆括号。用空格键选择相应的选项时会发现中括号中要么是空，要么是“*”；尖括号中可能是空、“*”和“M”，分别表示包含选项、不包含选项和编译成模块；圆括号的内容是要求用户在所提供的几个选项中选择一项。

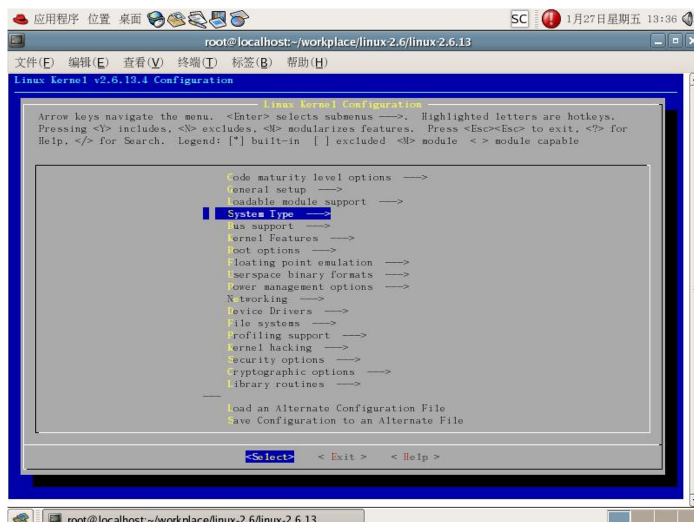


图 10.20　make menuconfig 配置界面

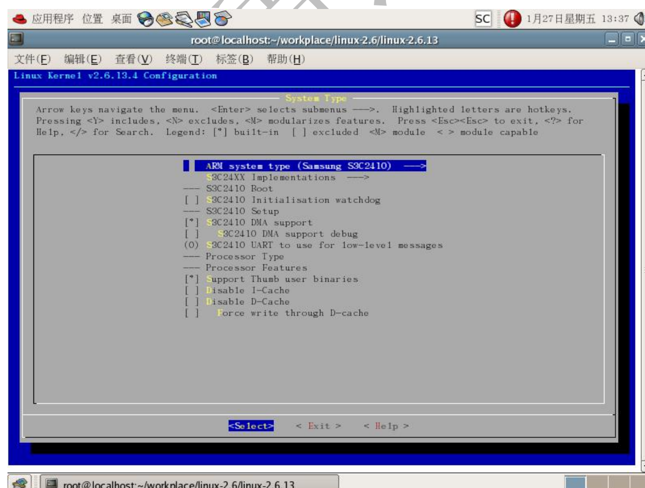


图 10.21　System Type 子项

此外，要注意 2.6 和 2.4 内核在串口命名上的一个重要区别，在 2.4 内核中“COM1”对应的是“ttyS0”，而在 2.6 内核中“COM1”对应的是“ttySAC0”，因此，在启动参数的子项要格外注意，如图 10.22 所示，否则串口打印不出信息。



一般情况下，使用厂商提供的默认配置文件都能正常运行，所以，用户初次使用时可以不用对其进行额外配置，以后需要其他功能时再另行添加，这样可以大大减少出错的几率，有利于错误定位。完成配置之后，就可以保存退出，如图 10.23 所示。

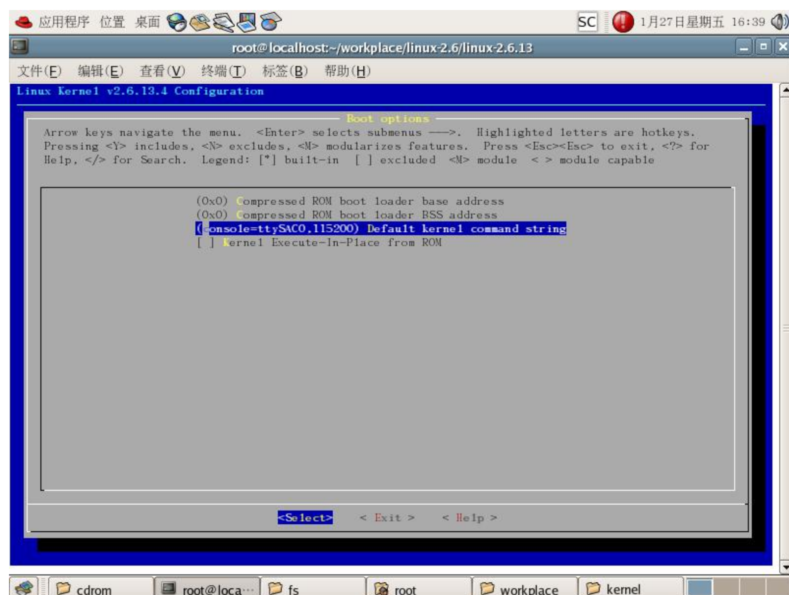


图 10.22　启动参数配置子项

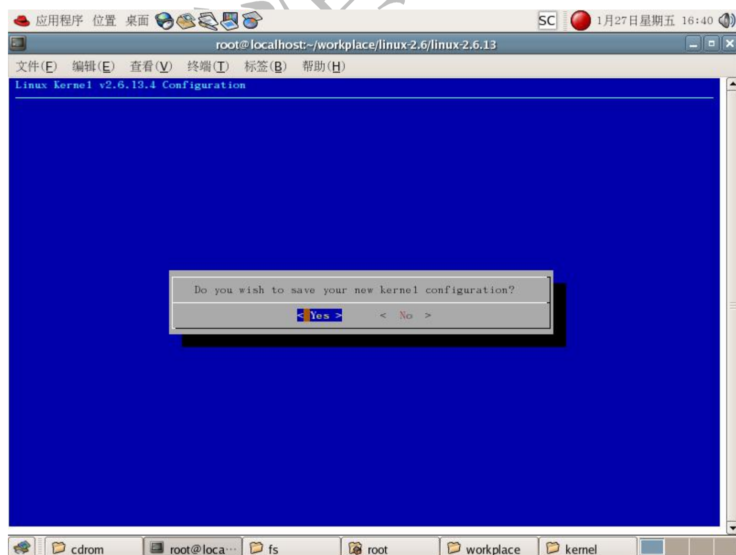


图 10.23　保存退出

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

2．建立依赖关系

由于内核源代码树中的大多数文件都与一些头文件有依赖关系，因此要顺利建立内核，内核源代码树中的每个 Makefile 就必须知道这些依赖关系。建立依赖关系往往发生在第一次编译内核的时候，它会在内核源代码树中每个子目录产生一个“.depend”文件。运行“make dep”即可。

3．建立内核

建立内核可以使用“make zImage”或“make bzImage”，这里建立的为压缩的内核映像。通常，在 Linux 中内核映像分为压缩的内核映像和未压缩的内核映像。其中，压缩的内核映像通常名称为 zImage，位于“arch/$（ARCH）/boot”目录中。而未压缩的内核映像通常名称为 vmlinux，位于源代码树的根目录中。

到这一步就完成了内核源代码的编译，之后，读者可以使用 10.1.3 节讲述的方法把内核压缩文件下载到开发板上运行。

10.1.5　Linux 内核目录结构

Linux 内核的目录结构如图 10.24 所示。

- /include 子目录包含建立内核代码时所需的大部分包含文件，这个模块利用其他模块重建内核。
- /init 子目录包含内核的初始化代码，这是内核工作的开始的起点。
- /arch 子目录包含所有硬件结构特定的内核代码。如 arm、i386、alpha。
- /drivers 子目录包含内核中所有的设备驱动程序，如块设备和 SCSI 设备。
- /fs 子目录包含所有的文件系统的代码，如 Ext2、VFAT 等。
- /net 子目录包含内核的连网代码。
- /mm 子目录包含所有内存管理代码。
- /ipc 子目录包含进程间通信代码。
- /kernel 子目录包含主内核代码。

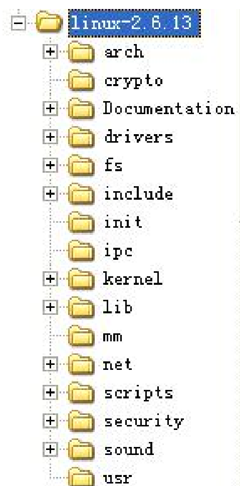


图 10.24　Linux 内核目录结构

10.1.6　制作文件系统

读者把 10.1.4 节中通过 make zImage 所编译的内核压缩映像下载到开发板后会发现，系统在进行了一些初始化的工作之后，并不能正常启动，如图 10.25 所示。

华清远见

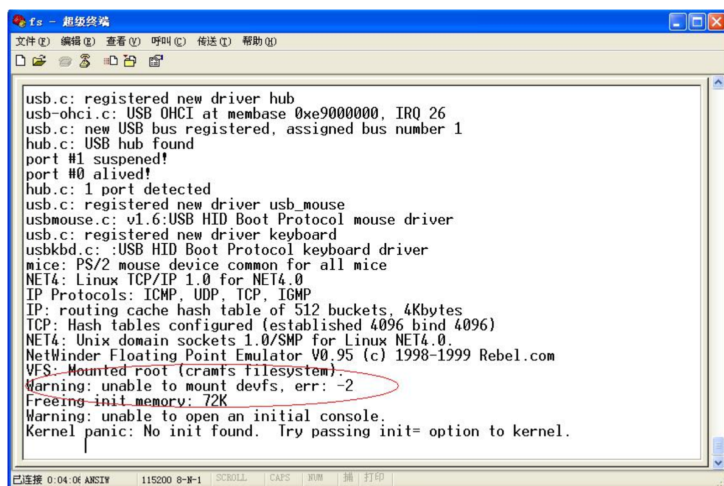


图 10.25 系统启动错误

可以看到，系统启动时发生了加载文件系统的错误。要记住，10.1.4 节所编译的仅仅是内核，文件系统和内核是完全独立的两个部分。读者可以回忆一下第 2 章讲解的 Linux 启动过程的分析（嵌入式 Linux 是 Linux 精简版本，其精髓部分是一样的），其中在 head.S 中就加载了根文件系统。因此，加载根文件系统是 Linux 启动中不可缺少的一部分。本节就来讲解嵌入式 Linux 中文件系统和根文件系统的制作方法。

制作文件系统的方法有很多，既可以从零开始手工制作，也可以在现有的基础上添加部分内容加载到目标板上去。由于完全手工制作工作量比较大，而且也很容易出错，因此，本节将主要介绍把现有的文件系统加载到目标板上的方法，主要包括制作文件系统镜像和用 NFS 加载文件系统的方法。

华清远见

1．制作文件系统镜像

读者已经知道，Linux 支持多种文件系统，同样，嵌入式 Linux 也支持多种文件系统。虽然在嵌入式中，由于资源受限的原因，它的文件系统和 Linux 的文件系统有较大的区别（前者往往是只读文件系统），但是，它们的总体架构是一样的，都是采用目录树的结构。在嵌入式中常见的文件系统有 cramfs、romfs、jffs、yaffs 等，这里就以制作 cramfs 文件系统为例进行讲解。cramfs 文件系统是一种经压缩的、极为简单的只读文件系统，因此非常适合嵌入式系统。要注意的是，不同的文件系统都有相应的制作工具，但是其主要的原理和制作方法是类似的。

制作 cramfs 文件系统需要用到的工具是 mkcramfs，下面就来介绍使用 mkcramfs 制作文件系统映像的方法。这里假设用户已经有了一个 cramfs 文件系统，在目录“/root/workplace/ fs/guo”中，如下所示：

```
[root@localhost guo]# ls
bin  dev  etc  home  lib  linuxrc  proc  Qtopia  ramdisk  sbin  tmp  usr  var
```

接下来就可以使用 mkcramfs 工具了，格式为：mkcramfs dir name，如下所示。

```
[root@localhost fs]# ./mkcramfs guo FS2410XP_camare_demo4.cramfs
-21.05% (-64 bytes)        Tongatapu
-21.03% (-49 bytes)       Truk
-21.03% (-49 bytes)       Wake
-22.41% (-52 bytes)       Wallis
-21.95% (-54 bytes)       Yap
-17.19% (-147 bytes)      WET
-47.88% (-8158 bytes)     zone.tab
-55.24% (-17421 bytes)    usb-storage.o
-54.18% (-16376 bytes)    usbvideo.o
-54.07% (-2736 bytes)     videodev.o
Everything: 27628 kilobytes
Super block: 76 bytes
CRC: e3a6d7ca
```

可以看到，mkcramfs 在制作文件镜像的时候对文件进行了压缩。

读者可以先在本机上通过 mount 进行验证，如下所示：

```
[root@localhost fs]# mkdir sunq
[root@localhost fs]# mount -o loop FS2410XP_camare_demo4.cramfs ./sunq
[root@localhost fs]# ls sunq
bin  dev  etc  home  lib  linuxrc  proc  Qtopia  ramdisk  sbin  tmp  usr  var
```

这时，就可以烧入到开发板的相应部分了。

2．NFS 文件系统

NFS 是 Network FileSystem 的简称，最早是由 Sun 公司提出并发展起来的，其目的就是让不同的机器、不同的操作系统之间可以彼此共享文件。NFS 可以让不同的主机通过网络将远端的 NFS 服务器共享出来的文件安装到自己的系统中，从客户端看来，使用 NFS 的远端文件就像是使用本地文件一样。在嵌入式中使用 NFS 会使应用程序的开发变得十分方便，并且不用反复地进行烧写镜像文件。

NFS 的使用分为服务器端和客户端，其中，服务器端是提供要共享的文件，而客户端则通过挂载“mount”这一动作来实现对共享文件的访问操作。下面主要介绍 NFS 服务器端的使用。

NFS 服务器端是通过读入它的配置文件“/etc/exports”来决定所共享的文件目录的。下面首先讲解这个配置文件的书写规范。

在这个配置文件中，每一行都代表一项要共享的文件目录及所指定的客户端对其的操作权限。客户端可以根据相应的权限，对该目录下的所有目录文件进行访问。配置文件中每一行的格式如下：

```
[共享的目录] [主机名称或 IP] [参数 1，参数 2…]
```

在这里，主机名或 IP 是可供共享的客户端主机名或 IP，若对所有的 IP 都可以访问，则可用“*”表示。

这里的参数有很多种组合方式，常见的参数如表 10.1 所示。

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com



表 10.1　常见参数

选　　项	参 数 含 义
rw	可读/写的权限
ro	只读的权限
no_root_squash	NFS 客户端分享目录使用者的权限，即如果客户端使用的是 root 用户，那么对于这个共享的目录而言，该客户端就具有 root 的权限
sync	资料同步写入到内存与硬盘当中
async	资料会先暂存于内存当中，而非直接写入硬盘

如在本例中，配置文件"/etc/exports"的代码如下：

```
[root@localhost fs]# cat /etc/exports
/root/workplace    *(rw,no_root_squash)
```

在设定完配置文件之后，需要启动 NFS 服务和 portmap 服务，这里的 portmap 服务是允许 NFS 客户端查看 NFS 服务所用的端口，在它被激活之后，就会出现一个端口号为 111 的 sun RPC（远端过程调用）的服务。这是 NFS 服务中必须实现的一项，因此，也必须把它开启。如下所示：

```
[root@localhost fs]# service portmap start
启动 portmap:                                          [确定]
[root@localhost fs]# service nfs start
启动 NFS 服务:                                         [确定]
关掉 NFS 配额:                                         [确定]
启动 NFS 守护进程:                                     [确定]
启动 NFS mountd:                                       [确定]
```

可以看到，在启动 NFS 服务时启动了 mountd 进程。这是 NFS 挂载服务，用于处理 NFSD 递交过来的客户端请求。另外，还会激活至少两个以上的系统守护进程，然后就开始监听客户端的请求，用 cat/var/log/messages 可以看到操作是否成功。这样，就启动了 NFS 的服务。另外还有两个命令，可以方便 NFS 的使用。

一个是 exportfs，它可以重新扫描"/etc/exports"，使用户在修改了"/etc/exports"配置文件后不需要每次都重启 NFS 服务。其格式为：

```
exportfs [选项]
```

exportfs 的常见选项如表 10.2 所示。

表 10.2　常见选项

选　　项	参 数 含 义
-a	全部挂载（或卸载）/etc/exports 中的设定文件目录
-r	重新挂载/etc/exports 中的设定文件目录
-u	卸载某一目录
-v	在 export 的时候，将共享的目录显示到屏幕上

另外一个是 showmount 命令，它用于当前的挂载情况。其格式如下：

华清远见 HQYJ.COM　华清远见教育集团官网：www.hqyj.com

```
showmount [选项] hostname
```

showmount 的常见选项如表 10.3 所示。

表 10.3　常见选项

选　　项	参 数 含 义
-a	在屏幕上显示目前主机与客户端所连上来的使用目录状态
-e	显示 hostname 中/etc/exports 里设定的共享目录

10.2 Bootloader 介绍



10.2.1　Bootloader 概述

1．概念

简单来说，Bootloader 就是在操作系统内核运行之前运行的一段程序，它类似于 PC 中的 BIOS 程序。通过这段程序，可以完成硬件设备的初始化，并建立内存空间的映射图的功能，从而将系统的软/硬件环境带到一个合适的状态，为最终调用系统内核做好准备。

通常，Bootloader 是严重地依赖于硬件实现的，特别是在嵌入式中。因此，在嵌入式世界中建立一个通用的 Bootloader 几乎是不可能的。尽管如此，仍然可以对 Bootloader 归纳出一些通用的概念来指导用户特定的 Bootloader 设计与实现。

（1）Bootloader 所支持的 CPU 和嵌入式开发板。每种不同的 CPU 体系结构都有不同的 Bootloader。有些 Bootloader 也支持多种体系结构的 CPU，如后面要介绍的 U-Boot 就同时支持 ARM 体系结构和 MIPS 体系结构。除了依赖于 CPU 的体系结构外，Bootloader 实际上也依赖于具体的嵌入式板级设备的配置。

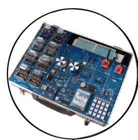
（2）Bootloader 的安装媒介。系统加电或复位后，所有的 CPU 通常都从某个由 CPU 制造商预先安排的地址上取指令。而基于 CPU 构建的嵌入式系统通常都有某种类型的固态存储设备（如 ROM、E^2PROM 或 Flash 等）被映射到这个预先安排的地址上。因此，在系统加电后，CPU 将首先执行 Bootloader 程序。

（3）Bootloader 的启动过程分为单阶段和多阶段两种。通常，多阶段的 Bootloader 能提供更为复杂的功能，以及更好的可移植性。

（4）Bootloader 的操作模式。大多数 Bootloader 都包含两种不同的操作模式：启动加载模式和下载模式，这种区别仅对于开发人员才有意义。

- 启动加载模式：这种模式也称为“自主”模式。也就是 Bootloader 从目标机上的某个固态存储设备上将操作系统加载到 RAM 中运行，整个过程并没有用户的介入。这种模式是嵌入式产品发布时的通用模式。

华清远见 HQYJ.COM　华清远见教育集团官网：www.hqyj.com



- 下载模式：在这种模式下，目标机上的 Bootloader 将通过串口连接或网络连接等通信手段从主机（Host）下载文件，例如，下载内核映像和根文件系统映像等。从主机下载的文件通常首先被 Bootloader 保存到目标机的 RAM 中，然后再被 Bootloader 写到目标机上的 Flash 类固态存储设备中。Bootloader 的这种模式系统在更新时使用。工作于这种模式下的 Bootloader 通常都会向它的终端用户提供一个简单的命令行接口。

（5）Bootloader 与主机之间进行文件传输所用的通信设备及协议，最常见的情况就是目标机上的 Bootloader 通过串口与主机之间进行文件传输，传输协议通常是 xmodem/ymodem/zmodem 协议中的一种。但是，串口传输的速度是有限的，因此，通过以太网连接并借助 TFTP 协议来下载文件是更好的选择。

2. Bootloader 启动流程

Bootloader 的启动流程一般分为两个阶段：Stage1 和 Stage2，下面分别对这两个阶段进行讲解。

1）Bootloader 的 Stage1

在 Stage1 中 Bootloader 主要完成以下工作。

- 基本的硬件初始化，包括屏蔽所有的中断、设置 CPU 的速度和时钟频率、RAM 初始化、初始化 LED、关闭 CPU 内部指令和数据 Cache 灯。
- 为加载 Stage2 准备 RAM 空间，通常为了获得更快的执行速度，通常把 Stage2 加载到 RAM 空间中来执行，因此，必须为加载 Bootloader 的 Stage2 准备好一段可用的 RAM 空间范围。
- 复制 Stage2 到 RAM 中，在这里要确定两点：①Stage2 的可执行映像在固态存储设备的存放起始地址和终止地址；②RAM 空间的起始地址。
- 设置堆栈指针 sp，这是为执行 Stage2 的 C 语言代码做好准备。

2）Bootloader 的 Stage2

在 Stage2 中 Bootloader 主要完成以下工作。

- 用汇编语言跳转到 main 入口函数。由于 Stage2 的代码通常用 C 语言来实现，目的是实现更复杂的功能和取得更好的代码可读性和可移植性。但是与普通 C 语言应用程序不同的是，在编译和链接 Bootloader 这样的程序时，不能使用 glibc 库中的任何支持函数。
- 初始化本阶段要使用到的硬件设备，包括初始化串口、初始化计时器等。在初始化这些设备之前，可以输出一些打印信息。
- 检测系统的内存映射。所谓内存映射就是指在整个 4GB 物理地址空间中指出哪些地址范围被分配用来寻址系统的 RAM 单元。
- 加载内核映像和根文件系统映像，这里包括规划内存占用的布局和从 Flash 中复制数据。

华清远见 HQYJ.COM 华清远见教育集团官网：www.hqyj.com

● 设置内核的启动参数。

3. Bootloader 的种类

嵌入式系统世界已经有各种各样的 Bootloader，种类划分也有多种方式。除了按照处理器体系结构不同划分以外，还有功能复杂程度的不同。

首先，区分一下“Bootloader”和“Monitor”的概念。严格来说，“Bootloader”只是引导设备并且执行主程序的固件；而“Monitor”还提供了更多的命令行接口，可以进行调试、读/写内存、烧写 Flash、配置环境变量等。“Monitor”在嵌入式系统开发过程中可以提供很好的调试功能，开发完成以后，就完全设置成了一个“Bootloader”。所以，习惯上大家把它们统称为 Bootloader。

表 10.4 列出了 Linux 的开放源代码引导程序及其支持的体系结构。表中给出了 X86、ARM、PowerPC 体系结构的常用引导程序，并且注明了每一种引导程序是否为“Monitor”。

表 10.4　开放源代码的 Linux 引导程序

Bootloader	Monitor	描　　述	X86	ARM	PowerPC
LILO	否	Linux 磁盘引导程序	是	否	否
GRUB	否	GNU 的 LILO 替代程序	是	否	否
Loadlin	否	从 DOS 引导 Linux	是	否	否
ROLO	否	从 ROM 引导 Linux 而不需要 BIOS	是	否	否
Etherboot	否	通过以太网卡启动 Linux 系统的固件	是	否	否
LinuxBIOS	否	完全替代 BOIS 的 Linux 引导程序	是	否	否
BLOB	否	LART 等硬件平台的引导程序	否	是	否
U-Boot	是	通用引导程序	是	是	是
RedBoot	是	基于 eCos 的引导程序	是	是	是

对于每种体系结构，都有一系列开放源代码 Bootloader 可以选用。

1）X86

X86 的工作站和服务器上一般使用 LILO 和 GRUB。LILO 是 Linux 发行版主流的 Bootloader。不过 Redhat Linux 发行版已经使用了 GRUB，GRUB 比 LILO 有更有好的显示接口，使用配置也更加灵活方便。

在某些 X86 嵌入式单板机或特殊设备上，会采用其他的 Bootloader，如 ROLO。这些 Bootloader 可以取代 BIOS 的功能，能够从 Flash 中直接引导 Linux 启动。现在 ROLO 支持的开发板已经并入 U-Boot，所以 U-Boot 也可以支持 X86 平台。

华清远见 HQYJ.COM　华清远见教育集团官网：www.hqyj.com

2）ARM

ARM 处理器的芯片商很多，所以每种芯片的开发板都有自己的 Bootloader。结果也使得 ARM Bootloader 也变得多种多样。最早有为 ARM720 处理器的开发板的固件，后来又有了 armboot、StrongARM 平台的 BLOB，以及 S3C2410 处理器开发板上的 vivi 等。现在 armboot 已经并入了 U-Boot，所以 U-Boot 也支持 ARM/XSCALE 平台。U-Boot 已经成为 ARM 平台事实上的标准 Bootloader。

3）PowerPC

PowerPC 平台的处理器有标准的 Bootloader，就是 PPCBOOT。PPCBOOT 在合并 armboot 等之后，创建了 U-Boot，成为各种体系结构开发板的通用引导程序。U-Boot 仍然是 PowerPC 平台的主要 Bootloader。

4）MIPS

MIPS 公司开发的 YAMON 是标准的 Bootloader，也有许多 MIPS 芯片商为自己的开发板写了 Bootloader。现在，U-Boot 也已经支持 MIPS 平台。

5）SH

SH 平台的标准 Bootloader 是 sh-boot。RedBoot 在这种平台上也很好用。

6）M68K

M68K 平台没有标准的 Bootloader。RedBoot 能够支持 M68K 系列的系统。

值得说明的是 RedBoot，它几乎能够支持所有的体系结构，包括 MIPS、SH、M68K 等。RedBoot 是以 eCos 为基础，采用 GPL 许可的开源软件工程。现在由 core eCos 的开发人员维护，源代码下载网站是 http://www.ecoscentric.com/snapshots。RedBoot 的文档也相当完善，有详细的使用手册 *RedBoot User's Guide*。

华清远见

10.2.2 U-Boot 概述

1. U-Boot 简介

U-Boot（Universal Bootloader）是遵循 GPL 条款的开放源代码项目。它是从 FADSROM、8xxROM、PPCBOOT 逐步发展演化而来。其源代码目录、编译形式与 Linux 内核很相似，事实上，不少 U-Boot 源代码就是相应的 Linux 内核源程序的简化，尤其是一些设备的驱动程序，从 U-Boot 源代码的注释中就能体现这一点。但是 U-Boot 不仅支持嵌入式 Linux 系统的引导，而且还支持 NetBSD、VxWorks、QNX、RTEMS、ARTOS、LynxOS 嵌入式操作系统。其目前要支持的目标操作系统有 OpenBSD、NetBSD、FreeBSD、4.4BSD、Linux、SVR4、Esix、Solaris、Irix、SCO、Dell、NCR、VxWorks、LynxOS、pSOS、QNX、RTEMS、ARTOS。这是 U-Boot 中 Universal 的一层含义，另外一层含义是 U-Boot 除了支持 PowerPC 系列的处理器外，还能支持 MIPS、X86、ARM、NIOS、XScale 等诸多常用系列的处理器。这两个特点正是 U-Boot 项目的开发目标，即

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

支持尽可能多的嵌入式处理器和嵌入式操作系统。到目前为止，U-Boot 对 PowerPC 系列处理器支持最为丰富，对 Linux 的支持最完善。

2．U-Boot 特点

U-Boot 的特点如下。

- 开放源代码。
- 支持多种嵌入式操作系统内核，如 Linux、NetBSD、VxWorks、QNX、RTEMS、ARTOS、LynxOS。
- 支持多个处理器系列，如 PowerPC、ARM、X86、MIPS、XScale。
- 较高的可靠性和稳定性。
- 高度灵活的功能设置，适合 U-Boot 调试，操作系统不同引导要求，产品发布等。
- 丰富的设备驱动源代码，如串口、以太网、SDRAM、Flash、LCD、NVRAM、E^2PROM、RTC、键盘等。
- 较为丰富的开发调试文档与强大的网络技术支持。

3．U-Boot 主要功能

U-Boot 可支持的主要功能如下。

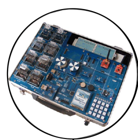
- 系统引导：支持 NFS 挂载、RAMDISK（压缩或非压缩）形式的根文件系统。支持 NFS 挂载，并从 Flash 中引导压缩或非压缩系统内核。
- 基本辅助功能：强大的操作系统接口功能；可灵活设置、传递多个关键参数给操作系统，适合系统在不同开发阶段的调试要求与产品发布，尤其对 Linux 支持最为强劲；支持目标板环境参数多种存储方式，如 Flash、NVRAM、E^2PROM；CRC32 校验，可校验 Flash 中内核、RAMDISK 镜像文件是否完好。
- 设备驱动：串口、SDRAM、Flash、以太网、LCD、NVRAM、E^2PROM、键盘、USB、PCMCIA、PCI、RTC 等驱动支持。
- 上电自检功能：SDRAM、Flash 大小自动检测；SDRAM 故障检测；CPU 型号。
- 特殊功能：XIP 内核引导。

4．U-Boot 的烧写

新开发的电路板没有任何程序可以执行，也就不能启动，需要先将 U-Boot 烧写到 Flash 中。如果主板上的 EPROM 或 Flash 能够取下来，就可以通过编程器烧写。例如，计算机 BIOS 就存储在一块 256KB 的 Flash 上，通过插座与主板连接。但是多数嵌入式单板使用贴片的 Flash，不能取下来烧写。这种情况可以通过处理器的调试接口，直接对板上的 Flash 编程。

处理器调试接口是为处理器芯片设计的标准调试接口，包含 BDM、JTAG 和 EJTAG 共 3 种接口标准。JTAG 接口已经介绍过；BDM（Background Debug Mode）主要应用在

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com



PowerPC8xx 系列处理器上；EJTAG 主要应用在 MIPS 处理器上。这 3 种硬件接口标准定义有所不同，但是功能基本相同，下面都统称为 JTAG 接口。

JTAG（Joint Test Action Group，联合测试行动小组）是一种国际标准测试协议（IEEE 1149.1 兼容），主要用于芯片内部测试。现在多数高级器件都支持 JTAG 协议，如 DSP、FPGA 器件等。标准的 JTAG 接口是 4 线，即 TMS、TCK、TDI、TDO，分别为模式选择、时钟、数据输入和数据输出线。JTAG 最初是用来对芯片进行测试的，基本原理是在器件内部定义一个 TAP（Test Access Port，测试访问口），通过专用的 JTAG 测试工具对内部节点进行测试。JTAG 测试允许多个器件通过 JTAG 接口串联在一起，形成一个 JTAG 链，能实现对各个器件分别测试。现在，JTAG 接口还常用于实现 ISP（In-System rogrammable，在线编程），对 Flash 等器件进行编程。JTAG 编程方式是在线编程，传统生产流程中先对芯片进行预编程再装到板上因此而改变，简化的流程为先固定器件到电路板上，再用 JTAG 编程，从而大大加快工程进度。JTAG 接口可对 PSD 芯片内部的所有部件进行编程。

JTAG 接口需要专用的硬件工具来连接。无论从功能、性能角度，还是从价格角度，这些工具都有很大差异。最简单的方式就是通过 JTAG 电缆，转接到计算机并口连接。这需要在主机端开发烧写程序，还需要有并口设备驱动程序。一个含有 JTAG Debug 接口模块的 CPU，只要时钟正常，就可以通过 JTAG 接口访问 CPU 的内部寄存器和挂在 CPU 总线上的设备，如 Flash、RAM、SoC（如 4510B、44B0X、AT91M 系列）内置模块的寄存器，定时器、GPIO 等寄存器。

开发板加电（或复位）时，烧写程序探测到处理器是否存在，并开始通信，然后把 Bootloader 下载并烧写到 Flash 中。这种方式速率很慢，平均每秒可以烧写 100～200B，不过价格却非常便宜。烧写完成后，复位实验板，串口终端应该显示 U-Boot 的启动信息。

5．U-Boot 的常用命令

U-Boot 上电启动后，按任意键可以退出自动启动状态，进入命令行，如下所示：

```
U-Boot 1.3.1 (Apr 26 2008 - 14:11:43)
U-Boot code: 11080000 -> 1109614C  BSS: -> 1109A91C
RAM Configuration:
Bank #0: 10000000 64 MB
Micron StrataFlash MT28F128J3 device initialized
Flash: 32 MB
In:    serial
Out:   serial
Err:   serial
Hit any key to stop autoboot:  0
U-Boot>
```

在命令行提示符下，可以输入 U-Boot 的命令并执行。U-Boot 可以支持几十个常用命令，通过这些命令，可以对开发板进行调试，可以引导 Linux 内核，还可以擦写 Flash 完成系统部署等功能。只有掌握这些命令，才能够顺利地进行嵌入式系统的开发。

华清远见 HQYJ.COM

华清远见教育集团官网：www.hqyj.com

输入 help 命令，可以得到当前 U-Boot 的所有命令列表。每一条命令后面是简单的命令说明。

```
?        - alias for 'help'
askenv   - get environment variables from stdin
autoscr  - run script from memory
base     - print or set address offset
bdinfo   - print Board Info structure
boot     - boot default, i.e., run 'bootcmd'
bootd    - boot default, i.e., run 'bootcmd'
bootelf  - Boot from an ELF image in memory
bootm    - boot application image from memory
bootp    - boot image via network using BootP/TFTP protocol
bootvx   - Boot vxWorks from an ELF image
cmp      - memory compare
coninfo  - print console devices and information
cp       - memory copy
crc32    - checksum calculation
date     - get/set/reset date & time
dcache   - enable or disable data cache
dhcp     - invoke DHCP client to obtain IP/boot params
echo     - echo args to console
erase    - erase Flash memory
fatinfo  - print information about filesystem
fatload  - load binary file from a dos filesystem
fatls    - list files in a directory (default /)
flinfo   - print Flash memory information
fsinfo   - print information about filesystems
fsload   - load binary file from a filesystem image
go       - start application at address 'addr'
help     - print online help
icache   - enable or disable instruction cache
iminfo   - print header information for application image
imls     - list all images found in Flash
itest    - return true/false on integer compare
loadb    - load binary file over serial line (kermit mode)
loads    - load S-Record file over serial line
loady    - load binary file over serial line (ymodem mode)
loop     - infinite loop on address range
ls       - list files in a directory (default /)
md       - memory display
mm       - memory modify (auto-incrementing)
mtest    - simple RAM test
mw       - memory write (fill)
nand     - legacy NAND sub-system
nboot    - boot from NAND device
nfs      - boot image via network using NFS protocol
nm       - memory modify (constant address)
ping     - send ICMP ECHO_REQUEST to network host
printenv - print environment variables
protect  - enable or disable Flash write protection
rarpboot - boot image via network using RARP/TFTP protocol
reset    - Perform RESET of the CPU
```

```
run      - run commands in an environment variable
saveenv  - save environment variables to persistent storage
setenv   - set environment variables
sleep    - delay execution for some time
tftpboot - boot image via network using TFTP protocol
usb      - USB sub-system
usbboot  - boot from USB device
version  - print monitor version
```

U-Boot 还提供了更加详细的命令帮助，通过 help 命令还可以查看每个命令的参数说明。由于开发过程的需要，有必要先把 U-Boot 命令的用法弄清楚。接下来，根据每一条命令的帮助信息，解释一下这些命令的功能和参数。

1）bootm 命令

bootm 命令可以引导启动存储在内存中的程序映像。这些内存包括 RAM 和可以永久保存的 Flash。

第 1 个参数 addr 是程序映像的地址，这个程序映像必须转换成 U-Boot 的格式。

第 2 个参数对于引导 Linux 内核有用，通常作为 U-Boot 格式的 RAMDISK 映像存储地址；也可以是传递给 Linux 内核的参数（默认情况下传递 bootargs 环境变量给内核）。

```
=> help bootm
bootm [addr [arg ...]]
    - boot application image stored in memory
         passing arguments 'arg ...'; when booting a Linux kernel,
         'arg' can be the address of an initrd image
```

2）bootp 命令

bootp 命令通过 bootp 请求，要求 DHCP 服务器分配 IP 地址，然后通过 TFTP 协议下载指定的文件到内存。

第 1 个参数是下载文件存放的内存地址。

第 2 个参数是要下载的文件名称，这个文件应该在开发主机上准备好。

```
=> help bootp
bootp [loadAddress] [bootfilename]
```

3）cmp 命令

cmp 命令可以比较两块内存中的内容。.b 以字节为单位；.w 以字为单位；.l 以长字为单位。注意：cmp.b 中间不能保留空格，需要连续输入命令。

第 1 个参数 addr1 是第一块内存的起始地址。

第 2 个参数 addr2 是第二块内存的起始地址。

第 3 个参数 count 是要比较的数目，单位是字节、字或长字。

```
=> help cmp
cmp [.b, .w, .l] addr1 addr2 count
     - compare memory
```


华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

4）cp 命令

cp 命令可以在内存中复制数据块，包括对 Flash 的读/写操作。

第 1 个参数 source 是要复制的数据块起始地址。

第 2 个参数 target 是数据块要复制到的地址。这个地址如果在 Flash 中，那么会直接调用写 Flash 的函数操作。所以，U-Boot 写 Flash 就使用这个命令，当然需要先把对应 Flash 区域擦干净。

第 3 个参数 count 是要复制的数目，根据 cp.b、cp.w、cp.l 分别以字节、字、长字为单位。

```
=> help cp
cp [.b, .w, .l] source target count
       - copy memory
```

5）crc32 命令

crc32 命令可以计算存储数据的校验和。

第 1 个参数 address 是需要校验的数据起始地址。

第 2 个参数 count 是要校验的数据字节数。

第 3 个参数 addr 用来指定保存结果的地址。

```
=> help crc32
crc32 address count [addr]
     - compute CRC32 checksum [save at addr]
```

6）echo 命令

echo 命令回显参数。

```
=> help echo
echo [args…]
      - echo args to console; \c suppresses newline
```

7）erase 命令

erase 命令可以擦除 Flash。参数必须指定 Flash 擦除的范围。

按照起始地址和结束地址，start 必须是擦除块的起始地址；end 必须是擦除末尾块的结束地址。这种方式最常用，例如，擦除 0x20000～0x3ffff 区域的命令为 erase 20000 3ffff。

按照组和扇区，*N* 表示 Flash 的组号，SF 表示擦除起始扇区号，SL 表示擦除结束扇区号。另外，还可以擦除整个组，擦除组号为 *N* 的整个 Flash 组。擦除全部 Flash 只要给出一个 all 的参数即可。

```
=> help erase
erase start end
      - erase Flash from addr 'start' to addr 'end'
erase N:SF[-SL]
      - erase sectors SF-SL in Flash bank # N
erase bank N
      - erase Flash bank # N
```


华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

```
erase all
     - erase all Flash banks
```

8）nand 命令

nand 命令可以通过不同的参数实现对 Nand Flash 的擦除、读、写操作。

常见的几种命令的含义如下（具体格式见 help nand）。

- nand erase：擦除 Nand Flash。
- nand read：读取 Nand Flash，遇到 Flash 坏块时会出错。
- nand read.jffs2：读取 Nand Flash，遇到坏块时会把坏块部分对应的内容填充为 0xff，不会出错。
- nand read.jffs2s：读取 Nand Flash，遇到坏块时自动跳过（建议使用）。
- nand write：写 Nand Flash，nand write 命令遇到 Flash 坏块时会出错。
- nand write.jffs2：写 Nand Flash，可自动跳过坏块（建议使用）。

```
=> help nand
nand info  - show available NAND devices
nand device [dev] - show or set current device
nand read[.jffs2[s]]  addr off size
nand write[.jffs2] addr off size - read/write 'size' bytes starting
   at offset 'off' to/from memory address `addr'
nand erase [clean] [off size] - erase 'size' bytes from
   offset 'off' (entire device if not specified)
nand bad - show bad blocks
nand read.oob addr off size - read out-of-band data
nand write.oob addr off size - read out-of-band data
```

9）flinfo 命令

flinfo 命令打印全部 Flash 组的信息，也可以只打印其中某个组。一般嵌入式系统的 Flash 只有一个组。

```
=> help flinfo
flinfo
      - print information for all Flash memory banks
flinfo N
      - print information for Flash memory bank # N
```

10）go 命令

go 命令可以执行应用程序。

第 1 个参数是要执行程序的入口地址。

第 2 个可选参数是传递给程序的参数，可以不用。

```
=> help go
go addr [arg ...]
     - start application at address 'addr'
       passing 'arg' as arguments
```

11）iminfo 命令

iminfo 可以打印程序映像的开头信息，包含映像内容的校验（序列号、头和校验和）。第 1 个参数指定映像的起始地址。可选的参数是指定更多的映像地址。

```
=> help iminfo
iminfo addr [addr ...]
     - print header information for application image starting at
       address 'addr' in memory; this includes verification of the
       image contents (magic number, header and payload checksums)
```

12）loadb 命令

loadb 命令可以通过串口线下载二进制格式文件。

```
=> help loadb
loadb [ off ] [ baud ]
    - load binary file over serial line with offset 'off' and baudrate 'baud'
```

13）loads 命令

loads 命令可以通过串口线下载 S-Record 格式文件。

```
=> help loads
loads [ off ]
    - load S-Record file over serial line with offset 'off'
```

14）mw 命令

mw 命令可以按照字节、字、长字写内存，.b、.w、.l 的用法与 cp 命令相同。

第 1 个参数 address 是要写的内存地址。

第 2 个参数 value 是要写的值。

第 3 个可选参数 count 是要写单位值的数目。

```
=> help mw
mw [.b, .w, .l] address value [count]
    - write memory
```

15）nfs 命令

nfs 命令可以使用 NFS 网络协议通过网络启动映像。

```
=> help nfs
nfs [loadAddress] [host ip addr:bootfilename]

=> help nm
nm [.b, .w, .l] address
    - memory modify, read and keep address
```

nm 命令可以修改内存，可以按照字节、字、长字操作。

参数 address 是要读出并且修改的内存地址。

16）printenv 命令

printenv 命令打印环境变量。可以打印全部环境变量，也可以只打印参数中列出的环境变量。

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

```
=> help printenv
printenv
      - print values of all environment variables
printenv name ...
      - print value of environment variable 'name'
```

17）protect 命令

protect 命令是对 Flash 写保护的操作，可以使能和解除写保护。

第 1 个参数 on 代表使能写保护；off 代表解除写保护。

第 2、第 3 个参数是指定 Flash 写保护操作范围，跟擦除的方式相同。

```
=> help protect
protect on  start end
      - protect Flash from addr 'start' to addr 'end'
protect on  N:SF[-SL]
      - protect sectors SF-SL in Flash bank # N
protect on  bank N
      - protect Flash bank # N
protect on  all
      - protect all Flash banks
protect off start end
      - make Flash from addr 'start' to addr 'end' writable
protect off N:SF[-SL]
     - make sectors SF-SL writable in Flash bank # N
protect off bank N
     - make Flash bank # N writable
protect off all
     - make all Flash banks writable
```

18）rarpboot 命令

rarpboot 命令可以使用 TFTP 协议通过网络启动映像。也就是把指定的文件下载到指定地址，然后执行。

第 1 个参数是映像文件下载到的内存地址。

第 2 个参数是要下载执行的镜像文件。

```
=> help rarpboot
rarpboot [loadAddress] [bootfilename]
```

19）run 命令

run 命令可以执行环境变量中的命令，后面的参数可以是几个环境变量名。

```
=> help run
run var [...]
      - run the commands in the environment variable(s) 'var'
```

20）setenv 命令

setenv 命令可以设置环境变量。

第 1 个参数是环境变量的名称。

第 2 个参数是要设置的值，如果没有第 2 个参数，表示删除这个环境变量。

```
=> help setenv
```


华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

```
setenv name value ...
     - set environment variable 'name' to 'value ...'
setenv name
     - delete environment variable 'name'
```

21）sleep 命令

tftpboot 命令可以使用 TFTP 协议通过网络下载文件。按照二进制文件格式下载。另外，使用这个命令，必须配置好相关的环境变量。例如，serverip 和 ipaddr。

第 1 个参数 loadAddress 是下载到的内存地址。

第 2 个参数是要下载的文件名称，必须放在 TFTP 服务器相应的目录下。

```
=> help sleep
sleep N
     - delay execution for N seconds (N is _decimal_ !!!)
```

sleep 命令可以延迟 *N* 秒执行，*N* 为十进制数。

```
=> help tftpboot
tftpboot [loadAddress] [bootfilename]
```

这些 U-Boot 命令为嵌入式系统提供了丰富的开发和调试功能。在 Linux 内核启动和调试过程中，都可以用到 U-Boot 的命令。但是一般情况下，不需要使用全部命令。例如，已经支持以太网接口，可以通过 tftpboot 命令来下载文件，那么还有必要使用串口下载的 loadb 吗？反过来，如果开发板需要特殊的调试功能，也可以添加新的命令。

10.3 嵌入式 Linux 根文件系统构建



10.3.1 根文件系统目录结构

文件系统是在任何操作系统中都非常重要的概念，简单来讲，文件系统是操作系统用于明确磁盘或分区上的文件的方法和数据结构，即在磁盘上组织文件的方法。文件系统的存在，使得数据可以被有效而透明地存取访问。

进行嵌入式开发，采用 Linux 作为嵌入式操作系统必须要对 Linux 文件系统结构有一定的了解。每个操作系统都有一种把数据保存为文件和目录的方法，因此它才能得知添加、修改之类的改变。在 DOS 操作系统之下，每个磁盘或磁盘分区有独立的根目录，并且用唯一的驱动器标识符来表示，如 C:\、D:\等。不同磁盘或不同的磁盘分区中，目录结构的根目录是各自独立的。而 Linux 的文件系统组织和 DOS 操作系统不同，它的文件系统是一个整体，所有的文件系统结合成一个完整的统一体，组织到一个树形目录结构之中，目录是树的枝干，这些目录可能会包含其他目录或其他目录的“父目录”，目录树的顶端是一个单独的根目录，用“/”表示。在 Linux 下可以看到系统的根目录组成内容，如图 10.26 所示。

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

```
root@zhang:/
文件(F)  编辑(E)  查看(V)  终端(T)  标签(B)  帮助(H)
[root@zhang mnt]# ls
[root@zhang mnt]# cd home
bash: cd: home: 没有那个文件或目录
[root@zhang mnt]# cd /home
[root@zhang home]# ls
win_c  win_d  zxq
[root@zhang home]# cd
[root@zhang ~]# ls
anaconda-ks.cfg                    libsigc++20-devel-2.0.6-1.i386.rpm
ccid-0.9.1                         libusb-0.1.8
Desktop                            makefile
glibmm24-2.4.5-1.i386.rpm          pcsc-lite-1.2.9
glibmm24-devel-2.4.5-1.i386.rpm    README
gtkmm24-2.4.8-1.i386.rpm           SCard.c
gtkmm24-devel-2.4.8-1.i386.rpm     Screenshot-1.png  根文件系统目录
gtkmm.tar.gz                       test
install.log                        test1
install.log.syslog                 tf2000v31u.tar.gz
libsigc++20-2.0.6-1.i386.rpm       tset
[root@zhang ~]# cd ..
[root@zhang /]# ls
bin   dev  home    lib         media  mnt  proc  sbin     srv  tftpboot  usr
boot  etc  initrd  lost+found  misc   opt  root  selinux  sys  tmp       var
[root@zhang /]#
```

图 10.26　Linux 下根目录内容

在图 10.26 中，椭圆框内的部分即为 Linux 根目录的组成。

10.3.2　FHS 目录结构

Linux 遵守文件系统科学分类标准（Filesystem Hierarchy Standard，FHS），FHS 是一个定义许多文件和目录的名字和位置的标准，该项标准可以在 http://www.pathname.com/FHS 找到，FHS 也是用来组织 Linux 和 UNIX 文件的方法，它使得 Linux 文件系统布局实现了标准化，一个 Linux 的根文件系统目录结构如图 10.27 所示。

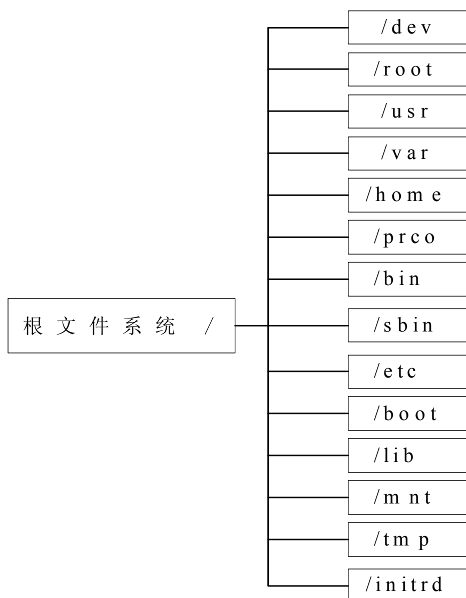


图 10.27　Linux 根文件系统结构

1．/dev 设备文件

在/dev 目录下是一些称为设备文件的特殊文件，用于访问系统资源或设备，如软盘、硬盘、系统内存等。设备文件的概念是 DOS 和 Windows 操作系统中所没有的，在 Linux 下，所有设备都被抽象成了文件，有了这些文件，用户可以像访问普通文件一样方便地访问系统中的物理设备。例如，可以像从一个文件中读取数据一样，通过读取/dev/mouse 文件用鼠标读取输入信息。在/dev 目录下，

每个文件都可以用 mknod 命令建立，各种设备所对应的特殊文件以一定规则来命名。以下是/dev 目录下的一些主要设备文件。

1）/dev/console

系统控制台，也就是直接和系统连接的监视器。

2）/dev/hd

在 Linux 系统中，对于 IDE 接口的整块硬盘表示为/dev/hd[a-z]，对于硬盘的不同分区，表示方法为/dev/hd[a-z]n，其中，n 表示的是该硬盘的不同分区情况。例如，/dev/hda 指的是第一个硬盘，hda1 则是指/dev/hda 的第一个分区。如系统中有其他的硬盘，则依次为/dev/hdb、/dev/hdc 等；如有多个分区，则依次为 hda1、hda2 等。

3）dev/fd

软驱设备文件。通过前面对系统 IDE 接口硬盘的表示方法不难理解：/dev/fd0 是指系统的第一个软驱，也就是通常所说的 A 盘，/dev/fd1 是指系统的第二个软驱。

4）dev/sd

SCSI 接口磁盘驱动器。理解方法和 IDE 接口的硬盘相同，只是把 hd 换成 sd。目前，Linux 下驱动 USB 存储设备的方法采用模拟 SCSI 设备，所以，USB 存储设备的表示方法与 SCSI 接口硬盘的表示方法相同。

5）dev/tty

设备虚拟控制台。如/dev/tty1 指的是系统的第一个虚拟控制台，/dev/tty2 则是系统的第二个虚拟控制台。

6）dev/ttyS*

串口设备文件。dev/ttyS0 是串口 1，dev/ttyS1 是串口 2。

2．/root root 用户主目录

root 目录中的内容包括引导系统的必备文件、文件系统的挂装信息、设备特殊文件，以及系统修复工具和备份工具等。由于是系统管理员的主目录，所以普通用户没有访问权限。

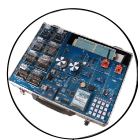
3．/usr

/usr 是最庞大的目录，该目录中包含一般不需要修改的命令程序文件、程序库、手册和其他文档等。Linux 内核的源代码就放在/usr/src/linux 中。

4．/var

该目录中包含经常变化的文件，例如，打印机、邮件、新闻等的脱机目录、日志文件，以及临时文件等。因为该文件系统的内容经常变化，所以如果和其他文件系统（如/usr）放在同一硬盘分区，文件系统的频繁变化将会提高整个文件系统的碎片化程度。

华清远见 HQYJ.COM 华清远见教育集团官网：www.hqyj.com



5．/home

用户主目录的默认位置。例如，一个名为 LY 的用户主目录将是/home/LY，系统的所有用户的数据保存在其主目录下。

6．/proc

需要注意的是，/prco 文件系统并不保存在系统的硬盘中，操作系统在内存中创建这一文件系统目录，是虚拟的目录，即系统内存的映射，其中包含一些和系统相关的信息，如 CPU 的信息等。

7．/bin

该目录包含二进制（Binary）文件的可执行程序，这里的 bin 本身就是 binary 的缩写，许多 Linux 命令就是放在该目录下的可执行程序，如 ls、mkdir、tar 等命令。

8．/sbin

与 bin 目录类似，存放系统编译后的可执行文件、命令，如常用到的 fsck、lsusb 等指令，通常只有 root 用户才有运行的权限。

9．/etc

/etc 目录在 Linux 文件系统中是一个很重要的目录，Linux 的很多系统配置文件就在该目录下，如系统初始化文件/etc/rc 等。Linux 正是靠这些文件才得以正常地运行，用户可以根据实际需要来配置相应的配置文件，下面列举一些配置文件。

1）/etc/rc 或/etc/rc.d

启动或改变运行级别时运行的脚本或脚本的目录。大多数的 Linux 发行版本中，启动脚本位于/etc/rc.d/init.d 中，系统最先运行的服务是那些放在/etc/rc.d 目录下的文件，而运行级别在文件/etc/inittab 中指定，这些会在后面的内容中详细讲到。

2）/etc/passwd

/etc/passwd 是存放用户的基本信息的口令文件。该口令文件的每一行都包含由 6 个冒号分隔的 7 个域，其中的域给出了用户名、真实姓名、用户起始目录、加密口令和用户的其他信息。

- username：用户名。
- passwd：是口令密文域。密文是加密过的口令。如果口令经过 shadow，则口令密文域只显示一个 x，通常，口令都应该经过 shadow 以确保安全。如果口令密文域显示为*，则表明该用户名有效但不能登录。如果口令密文域为空则表明该用户登录不需要口令。
- uid：系统用于唯一标识用户名的数字。
- gid：表示用户所在默认组号。
- comments：用户的个人信息。

华清远见

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

- directory：定义用户的初始工作目录。
- shell：指定用户登录到系统后启动的外壳程序。

3）etc/fstab

指定启动时需要自动安装的文件系统列表。通常如果用户在使用过程中需要手动加载许多文件系统，将会带来不小的工作量。为了避免这些麻烦，让系统在启动时自动加载这些文件系统，Linux 中使用/etc/fstab 文件来完成这一功能。fstab 文件中列出了引导时需安装的文件系统的类型、加载点及可选参数。所以，进行相应的配置即可确定系统引导时加载的文件系统。

4）etc/inittab

init 的配置文件，在后面将会详细讲解。

10．/boot

该目录存放系统启动时所需的各种文件，如内核的镜像文件、引导加载器（Bootstrap Loader）使用的文件 LILO 和 GRUB。

11．/lib

标准程序设计库，又称动态链接共享库，作用类似于 Windows 中的.dll 文件。

12．/mnt

该目录用来为其他文件系统提供安装点，例如，可以在该目下新建一个目录 floppy 用来挂载软盘，同样，可以新建一个目录 cdrom（可以用任意名称）用来挂载光盘等。例如，在 Linux 下的终端执行下面的语句：

```
# mount -t  vfat  dev/hda1  /mnt/win_D
```

即可将硬盘的第一个分区挂载到 Linux 下的/mnt/win_D 目录中。

13．/tmp

公用的临时文件存储点。

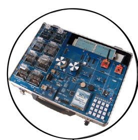
14．/initrd

用来在计算机启动时挂载 initrd.img 映像文件及载入所需设备模块的目录，需要注意的是，不要随便删除/initrd/目录，如果删除了该目录，将无法重新引导系统。

10.3.3 文件存放规则

为了实现各种 Linux 版本系统的标准化，各种不同的 Linux 版本都会根据 FHS（Filesystem Hierarchy Standard）标准来进行系统管理，这也使得 Linux 系统的兼容性大大提高。FHS 规定了两级目录，第一级是根目录下的主要目录，根据目录名称可以得知其中应该放置什么样的文件，如/etc 目录下应该放置各种配置文件，/bin 和/sbin 目录下

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com



应该放置相应的可执行文件等；第二级目录则主要针对/usr 和/var 做出了更深层目录的定义。

UNIX/Linux 系统很长时间以来一直是在“什么文件放在哪里”的基础之上建立文件存放规则的，并且按照这些规则把文件放进相应分级结构中。文件系统分级结构标准（FHS）试图以一种合乎逻辑的方式定义这些规则，而且在 Linux 上得到了广泛应用。按照 FHS 标准，在 Linux 下存放文件主要有以下一些规则。

- 把全局配置文件放入/etc 目录下。
- 将设备文件信息放入/dev 目录下，设备名可以作为符号链接定位在/dev 中或/dev 子目录中的其他设备存在。
- 操作系统核心定位在/或/boot，若操作系统核心不是作为文件系统的一个文件存在，不应用它。
- 库存放的目录是/lib。
- 存放系统编译后的可执行文件、命令的目录是/bin、/sbin、/usr。

10.4 本章习题



1．什么是交叉工具链？如何创建？
2．超级终端在嵌入式开发中起到的作用是什么？如何配置？
3．什么是 Bootloader？
4．什么是 U-Boot？简述其主要的目录结构。
5．如何编译 U-Boot？
6．如何构建一个根文件系统？如何测试文件系统的完整性？

华清远见

华清远见 HQYJ.COM

华清远见教育集团官网：www.hqyj.com

华清远见 HQYJ.COM

第 11 章 Android 系统的编译和移植

在第 10 章中介绍了如何构建一套嵌入式 Linux 开发环境，以及 Bootloader 和交叉工具链的概念用法，作为建立在 Linux 内核基础上的 Android 操作系统，它的编译和移植无论是过程还是技术上都和嵌入式 Linux 非常相似，所以本章将在第 10 章的基础上介绍一个典型的 Android 系统的编译和移植。

华清远见

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

11.1 移植背景与目标



现有的环境是一套能够正常运行 Linux-2.6.21 的 EZ6410（基于 S3C6410）硬件系统。移植目标是在 EZ6410 系统上运行 Android-2.3 系统。

11.2 移植涉及的主要过程



移植涉及的主要过程如下：

- 下载 Android Linux 内核。
- 安装交叉工具链。
- 移植 Android Linux 内核支持 EZ6410 平台。
- 安装 Android SDK。
- 获得 Android 根文件系统。
- 设置系统环境，完成 Android 正常启动。

11.3 下载 Android Linux 内核



目前支持 S3C6410 硬件的 Android 系统可以在网上找到。可以在 http://code.google.com/hosting/中搜索 S3C6410，看到一些支持 S3C6410 的 Android 项目，如图 11.1 所示。

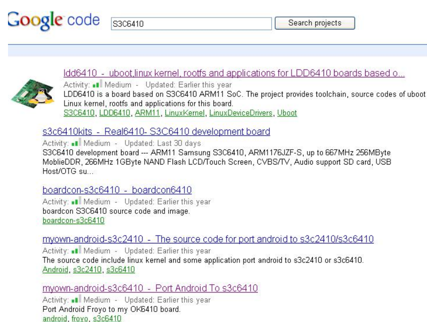


图 11.1　Android 系统下载界面

我们选择借鉴 ldd6410 项目（http://code.google.com/p/ldd6410/）。

LDD6410 的硬件结构如图 11.2 所示，我们需要针对其与 EZ6410 硬件结构的差异进行移植。EZ6410 的具体硬件配置参考开发板手册。

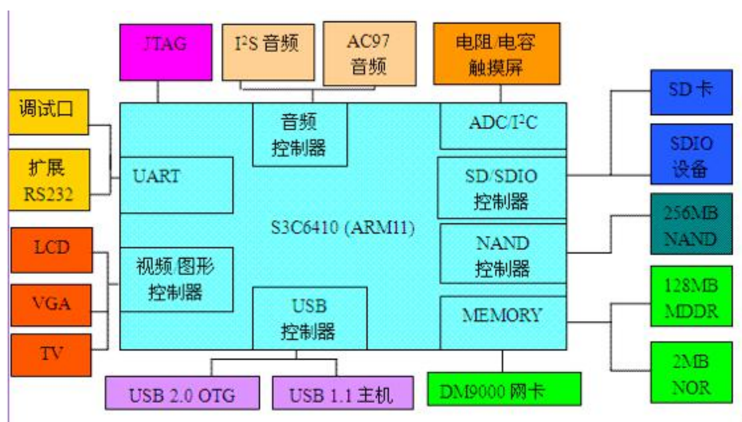


图 11.2　LDD6410 开发板结构图

使用下列指令下载：

```
# svn checkout http://ldd6410.googlecode.com/svn/trunk/ ldd6410-read-only
```

下载完成后，在 ldd6410-read-only 目录下有下载好的内核。

LDD6410 整合了完整的 Android 驱动（位于 drivers/android 下的 binder、lowmemorykiller 等）、内核电源管理（位于 kernel/power 下的 wakelock、userwakelock 等）、ashmem 补丁（位于 mm/ashmem.c）和虚拟电池（drivers/power/fake_battery.c）等。

如图 11.3 所示为 drivers/android 下驱动的配置。

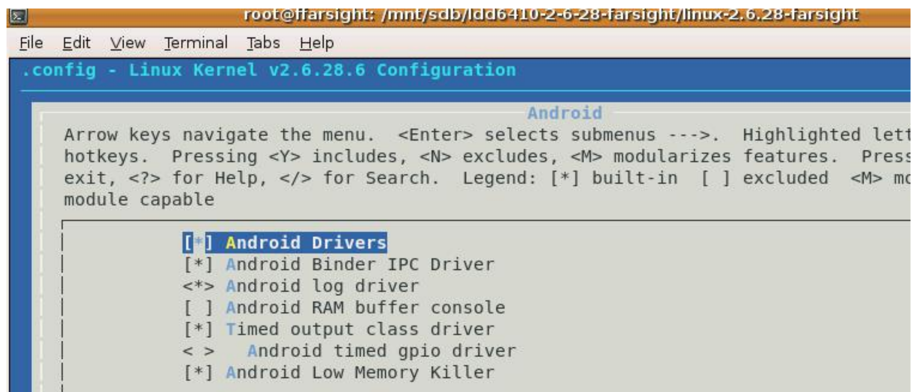


图 11.3　Android 驱动配置界面

如图 11.4 所示为 kernel/power 下 Android 电源管理的配置。



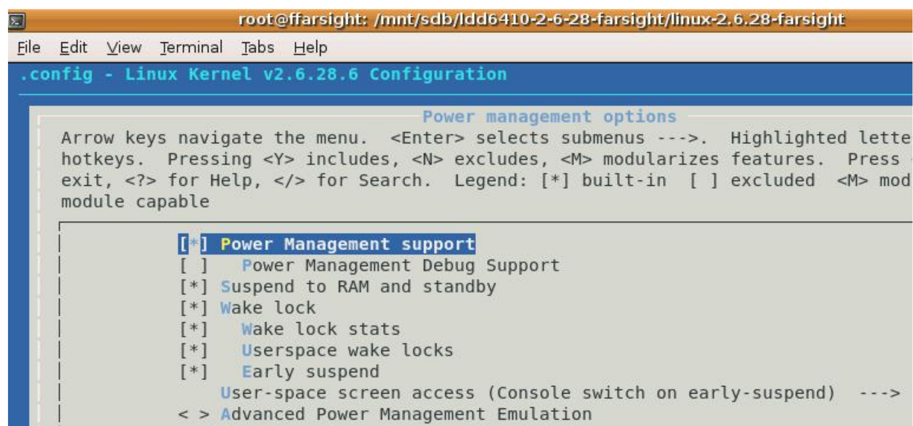


图 11.4 Android 系统电源管理界面

11.4 安装交叉工具链



在开始内核移植之前，先完成工具链的搭建。解压 EZ6410 光盘\B\Linux\cross_tools\arm-none-linux-gnueabi.tar.bz2 到 ubantu-8.10 系统的\usr\local\arm 目录下。

在 ldd6410-read-only 目录下，修改 vim .cross_compile 内容为：/usr/local/arm/arm-none-linux-gnueabi/bin/arm-none-linux-guneabi-。

11.5 Android Linux 内核支持 EZ6410 平台

移植过程中，发现硬件差异主要如下。

- 网卡：ldd6410 使用的网卡是 dm9000（Ldd6410→dm9000）而 EZ6410 使用的网卡是 CS8900a（EZ6410→CS8900a），所以我们需要移植 CS8900a 驱动。
- 键盘：键盘接线不一样，需要编写新的键盘驱动。
- 液晶、触摸屏：液晶的品牌、分辨率不一样，需要移植液晶、触摸屏驱动。
- USB 时钟系统：正确使用 USB 功能，需要修改 USB HOST 驱动的代码。

11.5.1 CS8900a 驱动移植

将实验代码目录下的 cs89x0.c 复制到内核的 drivers/net/目录下。配置内核支持 cs8900a 驱动，代码如下：

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

```
#vim Makefile
在 obj-$(CONFIG_DM9000) += dm9000.o 下添加
obj-$(CONFIG_CS89x0) += cs89x0.o
#vim Kconfig
tristate "CS89x0 support"
depends on NET_ETHERNET && (ISA || EISA || MACH_IXDP2351 \
       || ARCH_IXDP2X01 || ARCH_PNX010X || MACH_MX31ADS)
其中的 depends on 加上 ||ARCH_S3C64XX||MACH_6410))
```

配置内核支持网卡驱动界面如图 11.5 所示。

```
--- Ethernet (10 or 100Mbit)
-*-   Generic Media Independent Interface device support
< >   ASIX AX88796 NE2000 clone support
< >   SMC 91C9x/91C1xxx support
<*>   DM9000 support
(4)     DM9000 maximum debug level
[ ]     Force simple NSR based PHY polling
< >   SMSC LAN911[5678] support
< >   Broadcom 440x/47xx ethernet support
<*>   CS89x0 support
```

图 11.5　配置内核支持网卡驱动界面

11.5.2　键盘驱动编写

针对 Android 要求，编写键盘驱动。EZ6410 实验平台上有 8 个按键：K1~K8。

Linux 系统提供了 input 子系统，按键、触摸屏、键盘、鼠标等输入都可以利用 input 接口函数来实现设备驱动，因此，按键和触摸屏设备驱动都可以作为 input 设备驱动而实现。

键盘原理图如图 11.6 所示。

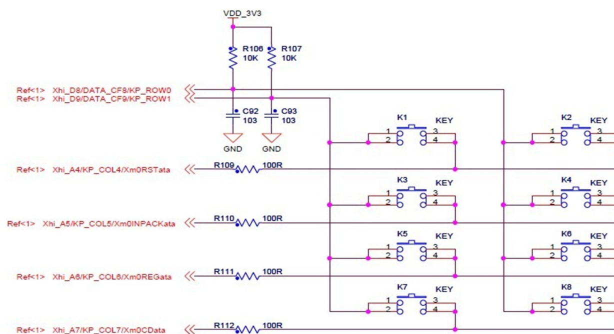


图 11.6　键盘原理图

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

Android 键值要求在 Android 的文件系统中的 system/usr/keylayout/qwerty.kl 中，代码如下：

```
#vim system/usr/keylayout/qwerty.kl
#define BACK            158
#define SOFT_RIGHT      60
#define MENU            229
#define SEARCH          127
#define HOME            102
#define DPAD_CENTER     232
#define DPAD_DOWN       108
#define DPAD_UP         103
#define DPAD_LEFT       105
#define DPAD_RIGHT      106
…
```

我们实现其中的 8 个按键，对应关系如下：

K1(RIGHT)	K2(CENTER)	K3(MENU)	K4(DOWN)
K5(HOME)	K6(UP)	K7(BACK)	K8(LEFT)

代码参见“实验代码\键盘驱动代码”目录下的 key_drv.c。

11.5.3 液晶驱动

修改 LDD6410 中的液晶驱动支持 EZ6410 平台的 320×240 的液晶屏。

修改的文件为 drivers/video/samsung/s3cfb_wanxin.c。

修改如下：

```
#elif defined (FB_320_240)
/* 320*240 */
#define S3CFB_HFP       30  /* front porch */
#define S3CFB_HSW       41 /* hsync width */
#define S3CFB_HBP       20 /* back porch */

#define S3CFB_VFP       15   /* front porch */
#define S3CFB_VSW       10   /* vsync width */
#define S3CFB_VBP       10  /* back porch */

#define S3CFB_HRES      320    /* horizon pixel  x resolition */
#define S3CFB_VRES      240 /* line cnt       y resolution */

#define S3CFB_HRES_VIRTUAL 320    /* horizon pixel  x resolition */
#define S3CFB_VRES_VIRTUAL (240*2)     /* line cnt       y resolution */

#define S3CFB_HRES_OSD     320    /* horizon pixel  x resolition */
#define S3CFB_VRES_OSD     240  /* line cnt       y resolution */
```

11.5.4　触摸屏驱动

将LDD6410系统移植到EZ6410开发板上后发现触摸屏在左右方向与系统的桌面应用相反。因此，必须修改内核触摸屏的驱动程序。触摸屏坐标界面如图11.7所示。

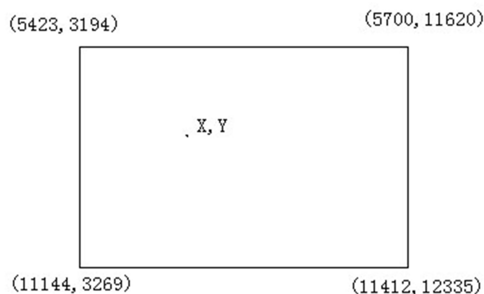


图11.7　触摸屏坐标界面

公式如下：

```
X:440=(ts->yp-3194):(11620-3194)
Y:272=(ts->xp-5423):(11144-5423)
```

修改触摸屏驱动drivers/input/touchscreen/s3c-ts.c，参见“触摸屏驱动”目录下的代码。

```
                    ts->yp /= 16; ts->xp /= 16;
                    x = (ts->yp-3194/16)*440/(11620/16-3194/16);
                    y = (ts->xp-5423/16)*272/(11144/16-5423/16);
                    if(x<0) x = 0; if(x>439) x = 439;
                    if(y<0) y = 0; if(y>271) y = 271;
//printk("x=%d,y=%d\n",x,y);
```

11.5.5　USB驱动修改

初始化没问题，但一插上USB就报如下错误：

```
/ # usb 1-1: new full speed USB device using s3c2410-ohci and address 2
usb 1-1: device descriptor read/64, error -62
usb 1-1: device descriptor read/64, error -62
usb 1-1: new full speed USB device using s3c2410-ohci and address 3
usb 1-1: device descriptor read/64, error -62
usb 1-1: device descriptor read/64, error -62
usb 1-1: new full speed USB device using s3c2410-ohci and address 4
usb 1-1: device not accepting address 4, error -62
usb 1-1: new full speed USB device using s3c2410-ohci and address 5
usb 1-1: device not accepting address 5, error -62
hub 1-0:1.0: unable to enumerate USB device on port 1
```


华清远见 HQYJ.COM　华清远见教育集团官网：www.hqyj.com



问题在 ohci-s3c2410.c 中，时钟设置出了问题。原来是 USB Host 的 48MHz 时钟没有起来。

S3C6410 支持 3 个 PLL，分别是 APLL、MPLL 和 EPLL。APLL 为 ARM 提供时钟，产生 ARMCLK，MPLL 为所有和 AXI/AHB/APB 相连的模块提供时钟，产生 HCLK 和 PCLK，EPLL 为特殊的外设提供时钟，产生 SCLK。如图 11.7 所示为 EPLL_CON 的 M、P 和 S 的取值。

FIN (MHz)	FOUT (MHz)	MDIV	PDIV	SDIV	KDIV
12	36	48	1	4	0
12	48	32	1	3	0
12	60	40	1	3	0

图 11.7　EPLL_CON 的 M、P 和 S 的取值

如图 11.8 所示为 USB 时钟原理图。

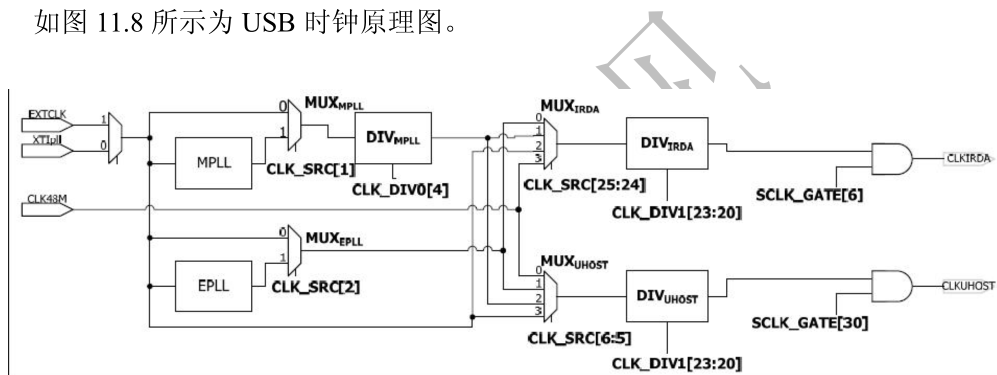


图 11.8　USB 时钟原理图

图 11.8 描述的是用于 IrDA 和 USB host 的时钟发生器，通常 USB 接口需要 48MHz 的操作时钟。

从图中可也以说明，HCLK_GATE、PCLK_GATE 和 SCL_GATE 控制时钟操作。如果一个未设置，则通过每个时钟分频器相应的时钟将会被提供，否则，将被屏蔽。

HCLK_GATE 控制 HCLK，用于每个 Ips。每个 IP 的 AHB 接口逻辑被独立地屏蔽，以减少动态电力消耗。PCLK_GATE 控制 PCLK。通过 SCLK_GATE 时钟被控制。

根据图 11.8 的 EPLL 通道写出以下程序：

```
#define EPLL_CON00     ((1<<31)|(0x20<<16)|(1<<8)|(3<<0))
#define EPLL_CON01          0
#define UPLL_SRC_MASK  ((1<<2)|(3<<5))
#define UPLL_SRC   ((1<<2)|(1<<5))
#define UPLL_DIV1_MASK      (0xf<<20)
#define UPLL_DIV1  (0<<20)
#define UPLL_GATE_MASK      (1<<30)
#define UPLL_GATE      (1<<30)
```


华清远见 HQYJ.COM　华清远见教育集团官网：www.hqyj.com

```
void set_upll(void)
{
    unsigned int tmp;

    while(__raw_readl(S3C_EPLL_CON0)!=EPLL_CON00)
    __raw_writel(EPLL_CON00,S3C_EPLL_CON0);
    while(__raw_readl(S3C_EPLL_CON1)!=EPLL_CON01)
    __raw_writel(EPLL_CON01,S3C_EPLL_CON1);
    while(((tmp= __raw_readl(S3C_CLK_SRC))&UPLL_SRC_MASK)!=UPLL_SRC)
        __raw_writel((tmp&UPLL_SRC_MASK)|UPLL_SRC,S3C_CLK_SRC);
    while(((tmp=__raw_readl(S3C_CLK_DIV1))&UPLL_DIV1_MASK)!=UPLL_DIV1)
        __raw_writel((tmp&UPLL_DIV1_MASK)|UPLL_DIV1,S3C_CLK_DIV1);
    while(((tmp=__raw_readl(S3C_SCLK_GATE))&UPLL_GATE_MASK)!=UPLL_GATE)
        __raw_writel((tmp&UPLL_GATE_MASK)|UPLL_GATE,S3C_SCLK_GATE);
}
```

在 probe 中加入上面的函数修改 USB host 的时钟：

```
set_upll();
```

然后编译内核，USB 的不能识别的错误就解决了。

参照 http://slash4.de/tutorials/Android_classes_-_Day_1_-_Installing_the_SDK 网页上的内容完成 SDK 的安装。

本例中安装的 SDK 包为 android-sdk-linux_x86-1.6_r1.tgz。

创建虚拟机平台 EZ6410，如图 11.9 所示。

```
# ./android create avd -n EZ6410 -t 2
```

```
root@ffarsight:/mnt/sdc/android-sdk-linux_x86-1.6_r1/tools# ./android create avd -n EZ6410 -t 2
Android 1.6 is a basic Android platform.
Do you wish to create a custom hardware profile [no]y

Device ram size: The amount of physical RAM on the device, in megabytes.
hw.ramSize [96]:128

Touch-screen support: Whether there is a touch screen or not on the device.
hw.touchScreen [yes]:

Track-ball support: Whether there is a trackball on the device.
hw.trackBall [yes]:no

Keyboard support: Whether the device has a QWERTY keyboard.
hw.keyboard [yes]:n

DPad support: Whether the device has DPad keys
hw.dPad [yes]:y

GSM modem support: Whether there is a GSM modem in the device.
hw.gsmModem [yes]:n

Camera support: Whether the device has a camera.
hw.camera [no]:n

Maximum horizontal camera pixels
hw.camera.maxHorizontalPixels [640]:
```

图 11.9　创建 EZ6410 虚拟平台界面

在主机上创建一个 sdcard image：

```
# sudo ./mksdcard 128M sdcard.img
```

启动 EZ6410 虚拟机：

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com




```
# sudo ./emulator -sdcard ./sdcard.img @EZ6410
```

Android 模拟器界面如图 11.10 所示。



图 11.10　Android 模拟器界面

11.5.6　提取 Android 根文件系统

连接虚拟机，使用如下指令：

```
#./adb shell
```

提取文件系统界面如图 11.11 所示。

```
root@ffarsight:/mnt/sdc/android-sdk-linux_x86-1.6_r1/tools# ./adb shell
* daemon not running. starting it now *
* daemon started successfully *
# mount
rootfs / rootfs ro 0 0
tmpfs /dev tmpfs rw,mode=755 0 0
devpts /dev/pts devpts rw,mode=600 0 0
proc /proc proc rw 0 0
sysfs /sys sysfs rw 0 0
tmpfs /sqlite_stmt_journals tmpfs rw,size=4096k 0 0
/dev/block/mtdblock0 /system yaffs2 ro 0 0
/dev/block/mtdblock1 /data yaffs2 rw,nosuid,nodev 0 0
/dev/block//vold/179:0 /sdcard vfat rw,dirsync,nosuid,nodev,noexec,uid=1000,gid=1015,fmask=0702,dmask=07
02,allow_utime=0020,codepage=cp437,iocharset=iso8859-1,shortname=mixed,utf8 0 0
# 
```

图 11.11　提取文件系统界面

把 busybox 放入模拟器目标机文件系统中：

```
root@ffarsight:/mnt/sdc/android-sdk-linux_x86-1.6_r1/tools# ./adb push /mnt/sdc/
busybox-1.16.1/busybox /data
466 KB/s (1875744 bytes in 3.925s)
```

把/system、/data、/sbin 目录及根目录下的 init、init.rc 等都放入 sdcard 的 image 中：

华清远见
HQYJ.COM
华清远见教育集团官网：www.hqyj.com

```
# ./busybox tar cvf /sdcard/android.tar /data /system /sbin /sqlite_stmt_journals /init.rc /init.goldfis
h.rc /init
```

查看结果，如图11.12所示。

```
# cd /sdcard
# ls
LOST.DIR
android.tar
#
```

图11.12　运行结果

在主机上以loop方式加载sdcard的image，并将里面的文件放到EZ6410的根文件系统：

```
root@ffarsight:/mnt/sdc/android-sdk-linux_x86-1.6_r1/tools# modprobe loop
root@ffarsight:/mnt/sdc/android-sdk-linux_x86-1.6_r1/tools# mount -o loop
sdcard.img /mnt/sd
root@ffarsight:/mnt/sdc/android-sdk-linux_x86-1.6_r1/tools# cd /mnt/sd
root@ffarsight:/mnt/sd# ls
android.tar  lost.dir
```

在原有的Linux的NFS文件系统目录下创建一个新的目录rootfs_test，并把android.tar解压到rootfs_test目录下。

```
# tar xvf android.tar -C /source/rootfs_android/rootfs_test/
```

在NFS服务目录/source/rootfs_android下添加一个文件Android.sh，如图11.13所示。

```
1 #!/bin/bash
2 echo "starting android ..."
3 umount /sys
4 #umount /dev/pts
5 umount /proc
6 umask 000
7 chroot /rootfs_test /system/bin/sh
8 #chroot /android_rootfs /system/bin/sh
9 #chroot /fs_donut_google_generic /system/bin/sh
"android.sh" 9 行 --11%--
```

图11.13　添加Android.sh文件界面

11.5.7　系统环境设置

复制image目录下的zImage到/tftpboot目录下。

解压android rootfs目录下的rootfs_android-farsight.tar.gz到/source/rootfs_android目录下。

华清远见 HQYJ.COM
华清远见教育集团官网：www.hqyj.com

设置好 NFS 环境，添加/source/rootfs_android 目录到 NFS 服务目录中。

设置 uboot 参数如下：

```
    SMDK6410 # print
    bootdelay=3
    baudrate=115200
    ethaddr=00:40:5c:26:0a:5b
    bootargs=root=nfs  nfsroot=192.168.1.10:/source/rootfs_android  ip=192.168.1.20
console=ttySAC0,115200
    filesize=1febe0
    fileaddr=C0008000
    gatewayip=192.168.1.1
    netmask=255.255.255.0
    ipaddr=192.168.1.20
    serverip=192.168.1.10
    bootcmd=tftp 0xc0008000 zImage ; bootm 0xc0008000
    stdin=serial
    stdout=serial
    stderr=serial

    Environment size: 366/16380 bytes
    SMDK6410 #
```

启动系统后，在串口终端中输入如下指令：

```
    [root@192 /]# ls
    android.sh   hh          linuxrc     root        tmp
    bin         home        mnt         rootfs_test  usr
    dev         key_drv.ko   opt          sbin        var
    etc         lib         proc        sys
    [root@192 /]# ./android.sh
    starting android ...
    # ./init
    init: cannot open '/initlogo.rle'
    sh: can't access tty; job control turned off
    # init: cannot find '/system/bin/playmp3', disabling 'bootsound'
    init: cannot find '/system/bin/dbus-daemon', disabling 'dbus'
    init: cannot find '/system/etc/install-recovery.sh', disabling 'flash_recovery'
    warning: `rild' uses 32-bit capabilities (legacy support in use)
    request_suspend_state:    wakeup    (3->0)    at    169456653237    (2030-09-10
04:05:05.064593851 UTC)
```

此时液晶屏上显示出和虚拟机一样的界面，可以按键进行操作。

如果想上网，可以按下面的方法配置 Android 网络上网。

在 Android 文件系统中配置网络，输入如下指令，用 Vim 打开 init.goldfish.sh 进行编辑：

```
    cd  system/etc/
    vim init.goldfish.sh
```

修改其中的网络配置：

```
            ifconfig eth0 10.0.2.15 netmask 255.255.255.0 up
            route add default gw 10.0.2.2 dev eth0
```

修改为：

```
        ifconfig eth0 192.168.1.12 netmask 255.255.255.0 up
        route add default gw 192.168.1.1 dev eth0
    setprop net.eth0.dns1 192.168.1.1
```

11.6 本章习题



1．Android 系统的移植主要分几个步骤？
2．怎么提取 Android 的根文件系统？
3．Android 系统的移植和嵌入式 Linux 的移植有什么相同和不同点？
4．怎么配置 Android 的交叉工具链？

华清远见